AFRL-RI-WP-TR-2008-1529

TERA-OP RELIABLE INTELLIGENTLY ADAPTIVE PROCESSING SYSTEM (TRIPS) IMPLEMENTATION

Stephen W. Keckler, Doug Burger, Kathryn S. McKinley, Steve Crago, and Richard Lethin

The University of Texas at Austin

SEPTEMBER 2008
Final Report

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Defense Advanced Research Projects Agency's Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-WP-TR-2008-1529 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH THE ASSIGNED DISTRIBUTION STATEMENT.

*//Signature//

ALFRED SCARPELLI
Project Engineer
Embedded Information Sys Engineering Branch
Advanced Computing Division

//Signature//

JAMES S. WILLIAMSON, Chief
Embedded Information Sys Engineering Branch
Advanced Computing Division

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
September 2008	Final	25 April 2003 – 30 September 2008

4. TITLE AND SUBTITLE
TERA-OP RELIABLE INTELLIGENTLY ADAPTIVE PROCESSING SYSTEM (TRIPS) IMPLEMENTATION

5a. CONTRACT NUMBER
F33615-03-C-4106

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62712E

6. AUTHOR(S)
Stephen W. Keckler, Doug Burger, Kathryn S. McKinley, Steve Crago, and Richard Lethin

5d. PROJECT NUMBER
P366

5e. TASK NUMBER
41

5f. WORK UNIT NUMBER
P3664106

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The University of Texas at Austin
Department of Computer Sciences
1 University Station, C0500
Austin, TX 78712-0233

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Information Directorate
Wright-Patterson Air Force Base, OH 45433-7334
Air Force Materiel Command
United States Air Force

DARPA/IPTO
3701 Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RITA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RI-WP-TR-2008-1529

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
PAO case number 12927, 03 Mar 2009. Document contains color.

14. ABSTRACT
The Tera-op Reliable Intelligently Adaptive Processing System (TRIPS) is a novel computer system designed to address technology scaling challenges, to provide high performance through concurrency, and to demonstrate mechanisms for hardware polymorphism. The team has constructed a full-system TRIPS prototype including a new Explicit Data Graph Execution (EDGE) instruction set architecture, custom application-specific integrated circuit (ASIC) chips, system circuit boards, a custom compiler with new optimization capabilities, a software development kit, and support for multithreaded parallel programs. Consisting of approximately 170 million transistors in a 130nm technology, the TRIPS chip includes two coarse grained processors, each with 16 ALUs (including floating-point units) that execute in parallel. TRIPS systems of up to 20 chips have been deployed at UT-Austin, ISI-East, and the Air Force Research Laboratory (AFRL). The prototype demonstrates per-processor performance (measured in cycles) of up to three times better than leading commercial products. Performance analysis results have led to follow-on architectures that employ dynamic polymorphism to tailor the capabilities of the hardware to the demands of the software.

15. SUBJECT TERMS
Scalable Architectures, Compilers, Polymorphous Computing, Parallel Software

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Al Scarpelli
Unclassified	Unclassified	Unclassified	SAR	112	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Table of Contents

List of Figures

List of Tables

Acknowledgments

The authors gratefully acknowledge the support of DARPA/IPTO and the Air Force Research Laboratory under contract F33615-03-C-4106.

In addition, all of the members of the TRIPS team made substantial and crucial contributions to the success of this project. These contributors include the scientists, students, and staff members at UT-Austin, IBM Microelectronics, the Information Sciences Institute of the University of Southern California, and Reservoir Laboratories.

1 Executive Summary

Semiconductor technology trends are presenting substantial challenges to the design and implementation of computer systems. Power and scaling constraints have brought an end to the continuous improvements of conventional uniprocessor designs. Further, power requirements in many deployment domains demand performance and power efficiencies unattainable by conventional general purpose processors. Polymorphous Computing Architectures (PCAs), architectures that can adapt to application and environmental requirements, are an attractive option for deployment in high-performance embedded systems. The present solution, in the absence of effective polymorphous systems, is to deploy multiple computers, with each one tailored to a particular anticipated task or class of tasks. While replacing many systems with a single one is a worthy goal, to date no suitable polymorphous systems exist.

In Phase 1 of the DARPA PCA program, completed in 2004, the Tera-Op Reliable Intelligently Adaptive Processing System (TRIPS) project developed and evaluated major advances in processor and memory system architectures that are scalable to deep-submicron fabrication technology and are adaptable to application demands. Phase 2 (TRIPS Implementation, described in this report), had four primary objectives: (1) implement a TRIPS hardware prototype to demonstrate capabilities of scalable and morphable systems, (2) implement a software development kit (SDK) for both internal and external users to port, author, and debug applications for TRIPS, (3) evaluate the TRIPS system and compare to existing commercial alternatives, and (4) develop and evaluate new architecture concepts based on the lessons learned from the prototype evaluation. The TRIPS team, composed of personnel from the University of Texas at Austin, USC Information Sciences Institute, Reservoir Laboratories, and IBM Microelectronics has reached each of these objectives.

The TRIPS hardware prototype design includes a custom ASIC consisting of approximately 170 million transistors in a $336mm^2$ chip fabricated in IBM's Cu-11 technology, a 130nm bulk technology with copper wiring. The TRIPS chip includes two coarse grained processors, each with 16 ALUs (including floating-point units) that execute in parallel. The processors execute an instruction set belonging to a new class of instruction set architectures (ISAs) called Explicit Data Graph Execution (EDGE). The chip also includes an implementation of a Non-Uniform Cache Architecture (NUCA), which is also designed for large, scalable, and configurable caches. TRIPS chips are assembled onto custom circuit boards with prototype system sizes ranging up to 32 chips (which includes a total of 64 processors and 1024 floating-point units). TRIPS systems of up to 20 chips have been deployed at UT-Austin, ISI-East, and AFRL. The software development kit (SDK) includes a custom optimizing compiler, a debugger, performance analysis tools, and support for multithreaded parallel programs. The compiler includes many new algorithms required for EDGE architectures, including optimizing instruction placement to reduce inter-instruction communication delays.

The evaluation of the TRIPS prototype system demonstrates per-processor performance (measured in cycles) of up to three times better than leading commercial products. Furthermore, the EDGE instruction set and execution model eliminates many of the power-hungry structures and operations in conventional computer systems. At 366MHz, a TRIPS processor consumes approximately 10 Watts, which gives the prototype a potential power efficiency advantage of a factor of three over conventional designs, with the opportunity for substantially more power optimizations beyond the scope of the prototype. On a synthetic aperture radar (SAR) benchmark, TRIPS outperformed conventional architectures by a factor of 1.5–1.8, depending on program phase. Based on measurements using the prototype, the team has developed both architecture optimizations to TRIPS and new composable core architectures that can adjust the processor granularity on the fly to meet the instantaneous parallelism demands of an application. This configurable lightweight processor (CLP) system is a substantial step forward in meeting the goals of truly polymorphous hardware systems.

The technical and educational impact of the TRIPS project has been profound, with scores of papers appearing in premier journals and conferences, best paper awards, numerous citations in papers authored by other researchers, and several projects at other universities derived directly from TRIPS observations and inventions. Concepts originating from the TRIPS non-uniform cache architectures (NUCA) for both single and multicore chips are making their way into the memory systems of commercial designs; we are likely to see products with NUCA-oriented caches in the near future. To date, the influence of the TRIPS processor technologies on commercial or COTS products have been more muted. Because TRIPS relies on a new instruction set architecture, convincing companies to adopt the TRIPS processor designs has been more difficult. At present, nearly all commercial systems are investing heavily in multicore system designs, despite the uncertainty in whether software can be developed to exploit the parallel cores. We anticipate that difficulties with that software transition will motivate a closer look in the future at the scalable processor architectures of TRIPS.

2 Introduction

TRIPS was conceived to solve many of the fundamental problems arising as semiconductor processes advance. Left unaddressed, these problems will inhibit the effective use of commercial components in military systems [1, 26]. Mission-critical applications will benefit from the advanced polymorphous capabilities of the TRIPS system, which include numerous malleable hardware components for ultra-flexible application mapping, resource oblivious scheduling for rapid application prototyping and deployment, environmental adaptivity for resilience in hostile environments, and dynamic performance adaptation for changing and unpredictable field constraints. The TRIPS system is intended to provide a single interface to applications that will allow them to run on a wide spectrum of SWEPT (Size, Weight, Energy, Power, and Time) implementations, including lightweight, battery-operated field systems, resilient but powerful base station systems, and ultra-high-performance back-room systems. Most important, TRIPS is intended to be extremely scalable with technology, and addresses many of the long-term problems that will emerge as devices scale down to 35 nanometers and below.

Phase 1 of the TRIPS project, concluded in 2004, developed proof-of-concept technologies that met the scalability and adaptivity goals described above [29]. To that end, the TRIPS team developed novel processor microarchitectures and on-chip memory system architectures that are both technology scalable and amenable to different types of applications. The processor core design employs an Explicit Data Graph Execution (EDGE) instruction set that enables different types of programs to be efficiently mapped to a spatial computation substrate. For the proof of concept, we developed architecture specifications, simulators, and prototype compilers to evaluate and refine the architecture concepts.

Phase 2 (described in this report), had four primary objectives intended to determine the feasibility and future of the technology developed in phase one: (1) implement a TRIPS hardware prototype to demonstrate capabilities of scalable and morphable systems, (2) implement a software development kit for both internal and external users to port, author, and debug applications for TRIPS, (3) evaluate the TRIPS system and compare to existing commercial alternatives, and (4) develop and evaluate new architecture concepts based on the lessons learned from the prototype evaluation. The TRIPS team, composed of personnel from the University of Texas at Austin, USC Information Sciences Institute, Reservoir Laboratories, and IBM Microelectronics has reached each of these objectives.

The TRIPS hardware prototype design includes a custom ASIC chip consisting of approximately 170 million transistors in a 130nm IBM fabrication technology. The TRIPS chip contains two coarse grained processors that execute in parallel, each with 16 ALUs (including floating-point units). The processors execute a new EDGE instruction set architecture designed for the distributed arrays of ALUs. The chip also includes an implementation of a Non-Uniform Cache Architecture, which is also designed for large scalable and configurable caches. TRIPS chips are assembled onto custom circuit boards with prototype system sizes ranging up to 32 chips (which includes a total of 64 processors and 1024 floating-point units). TRIPS systems of up to 20 chips have been deployed at UT-Austin, ISI-East, and AFRL. The SDK includes a custom optimizing compiler, a debugger, performance analysis tools, and support for multithreaded parallel programs. The compiler includes many new algorithms required for EDGE architectures, including optimizing instruction placement to reduce inter-instruction communication delays.

The evaluation of the TRIPS prototype system demonstrates per-processor performance (measured in cycles) of up to 3 times better than leading commercial products. Furthermore, the EDGE instruction set and execution model eliminates many of the power-hungry structures and operations in conventional computer systems including register renaming, instruction reordering, and register file reads and writes. At 366MHz, a TRIPS processor consumes approximately 10 Watts, which gives the prototype a potential power efficiency advantage of a factor of three over conventional designs, with the opportunity for substantially more power optimizations beyond the scope of the prototype. On a synthetic aperture radar (SAR) code, TRIPS outperformed conventional architectures by a factor of 1.5–1.8, depending on program phase. Based on observations and measurements with the prototype, the team has developed both architecture optimizations to TRIPS, as well as a new composable core architecture that can adjust the processor granularity on the fly to meet the instantaneous parallelism demands of an application. This configurable lightweight processor (CLP) system represents a substantial step forward in meeting the goals of truly polymorphous hardware systems.

The technical and educational impact of the TRIPS project has been profound. From the academic perspective, the TRIPS project has resulted in more than 11 journal articles or book chapters, more than 40 refereed papers presented at leading

technical conferences, more than 20 workshop papers, and 4 best paper awards. Just since 2002, these papers have garnered more than 1500 citations by other technical papers, demonstrating a dramatic influence on computer architecture research. Numerous follow-on research papers have been published based on our foundational work on NUCA. Our work has also inspired new research efforts in data-driven computer architectures. For example, the TRIPS software development kit was made available to other researchers upon request; to date, over 65 different researchers or research groups have obtained the tools.

Ten of the graduate students who have worked on TRIPS have graduated with doctoral degrees and some have taken their experience and insight to jobs at IBM, Intel, and AMD. Three of the graduates are on the faculty at leading universities including the University of Wisconsin, Columbia University, and Texas A&M University. More than twelve undergraduate students have worked on aspects of TRIPS, and many of these have gone on to pursue graduate degrees in computer science at leading U.S. universities.

To date, the influence of the TRIPS technologies on commercial or COTS products have been more muted. Concepts originating from the TRIPS NUCA for both single and multicore chips is making its way into the memory systems of commercial designs and we are likely to see products with NUCA-oriented caches in the near future. However, because TRIPS relies on a redesign of the instruction set architecture, convincing companies to adopt the TRIPS processor designs has been more difficult. At present, nearly all commercial systems are investing heavily in multicore system designs, despite the uncertainty in whether software can be developed to exploit the parallel cores. We anticipate that the difficulty associated with that software transition will motivate a closer look in the future at the scalable processor architectures of TRIPS.

Section 3 describes the TRIPS EDGE instruction set architecture in detail, including refinements made during the design of the TRIPS mircroarchitecture in phase two. Sections 4 and 5 detail the TRIPS prototype hardware design and implementation, including the TRIPS chips, boards, and systems. Sections 6–8 describe the TRIPS compiler, including the novel algorithms required for distributed processor architectures, and the tools that compose the software development kit (SDK). Section 9 recounts a performance evaluation of TRIPS, focusing primarily on the novel elements of the TRIPS processor and memory systems. Section 10 gives an overview of potential next generation polymorphous computing systems that consist of computational and memory elements that can be assembled dynamically to form logical processors, conforming to the concurrency needs of an applications. Section 11 provides a summary and Section 12 give recommendations for further study and use of TRIPS technologies.

3 EDGE Architectures

3.1 Background

The original first goal of the TRIPS project was to develop several new technologies for scalable processor and memory system architectures. Given the massive investment in conventional RISC and CISC architectures, and the attendant difficulties in parallelizing them for many applications. we developed a new class of instruction set architectures, called Explicit Data Graph Execution architectures. This new class of architectures addresses the scalability of the microprocessor core, permitting polymorphism, high performance, and high energy efficiency.

3.2 Explicit Data Graph Execution (EDGE) ISAs

EDGE ISAs combine the capabilities of the classic dataflow architectures with the support required to execute conventional imperative languages, such as C, Fortran, Java, or C++. They support design-productive tiled microarchitectures, wide issue out-of-order execution, and the exposure of large regions of computation to the hardware without the attendant circuit complexity of conventional superscalar processors that employ RISC or CISC ISAs.

EDGE ISAs have two main characteristics: (1) a program is partitioned into atomically executed blocks of instructions which have a single entry-point and no internal branching, and (2) instructions within a block execute in dataflow order according to their true data dependences. These two features permit efficient and polymorphous execution on a microarchitecture that supports them. In the case of the TRIPS prototype, the TRIPS chip fetches and executes individual blocks sequentially, but as an optimization the chip speculatively fetches and executes multiple subsequent blocks concurrently with the non-speculative block. In the TRIPS chip prototype, this method opens up an instruction window of up to 1024 instructions, providing the opportunity to exploit concurrency from different regions of the executing program.

3.2.1 Block-atomic Execution

Similar to the concept of transactions (although not visible to the programmer), an EDGE ISA specifies that its instruction blocks must commit atomically and in program order. Each block, which in the TRIPS system can contain up to 128 normal instructions, is fetched as a single logical unit, mapped to the hardware, and, when complete, is committed as one unit, with all of its outputs (stores, register writes, and one branch) logically committed to memory at once. This block-atomic model greatly reduces the overheads for supporting high-performance execution, such as wide instruction fetch and branch prediction. It also supports distributed execution by reducing the frequency at which control decisions must be made. Finally, having atomically committing instruction blocks that are of finite size reduces the scope of the dataflow execution, bounding the size of the instruction pointers required for the explicit data graph execution, as described below.

While we are not the first to discuss the notion of atomic instruction blocks, the combination of unusually large atomic blocks with dataflow semantics, as described below, creates a powerful capability not previously seen in computer architecture research.

3.2.2 Direct Instruction Communication with Explicit Targets

Within each block, execution proceeds directly from instruction to instruction, rather than through a shared register file. For non-memory instructions, producer-consumer relationships are explicit, with each instruction containing the instruction numbers of the instructions within a block that consumes its result (a “push” model). Conventional RISC and CISC ISAs specify their source operands as register names in the instructions, and are thus a “pull” model. When each EDGE instruction executes, it sends its result to its consumers, waking them up for execution as necessary. This model thus effectively encodes a statically formed dataflow graph, expressed through the ISA, to the hardware for each block.

The hardware maps these static graphs onto the execution units, and typically executes them in dataflow order. Each block reads from a number of architectural registers to commence its execution, and writes to a number of registers to communicate to other subsequent blocks. By stitching together a number of these blocks at runtime, a microarchitecture can create a larger instruction window, with the dynamically tracked dataflow arcs found at runtime through store/load pairs and inter-block register communication. This model eliminates a majority of the accesses to the shared register file, and in the common case of one ALU operation communicating with another, removes a significant amount of the execution energy associated with out-of-order execution. Sequential memory semantics are still obeyed, since the hardware must produce the same result as if each instruction was executed in the original program order. Since there is no ordering of non-memory instructions within a block other than dataflow order, each load and store contains a load/store identifier to convey its original program order to the memory system, which then guarantees correct execution.

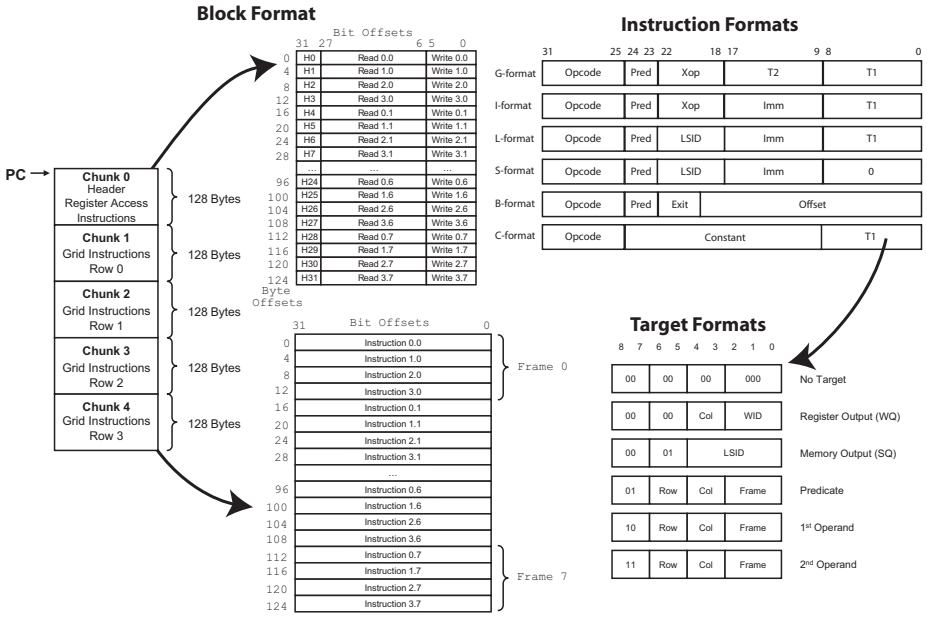
3.3 TRIPS ISA Specification

The TRIPS prototype instruction set is the first instantiation of an EDGE architecture. Figure 1 shows the current definition of the TRIPS prototype's ISA. Each block contains a 128-byte block header, which specifies the number of block outputs, as well as the registers that are read from and written to by the block. The rest of the block consists of four 128-byte instruction "chunks," each of which supplies 32 32-bit instructions to a row. Each row contains 4 nodes with 8 instructions per node.

The ISA provides the six instruction formats shown in Figure 1, which differ based on instruction function and number of distinct instruction targets. The major difference between the TRIPS EDGE instructions and conventional RISC instructions is that each instruction contains no source operand specifiers, but instead contains the name(s) of its consumer instructions. These consumer names are called targets; one or two per value-producing instruction are provided. If an instruction has more targets in a block than can be specified in the instruction format, extra instructions must be inserted to forward the value to the additional targets, resulting in a software fan-out tree. This "push" model permits the compiler to schedule dataflow-like execution within a block on the underlying substrate. Communication among different blocks must still occur through registers. As shown in Figure 1, targets in the TRIPS ISA consist of nine bits. Seven of the bits specify the target instruction number, which corresponds either to a physical reservation station in the execution array, or as a block output to one of the 128 architectural registers. Physical reservation stations are identified using row, column, and frame coordinates. The frame represents a specific buffer location at the ALU indicated by $< row, column >$. A 4×4 TRIPS processor with 128 reservation stations (one reservation station for each instruction in the block) requires 8 frames. The remaining two bits specify whether each target is an architectural register or is in the block, and if it is in the block, whether it is the left, right, or predicate operand of the consuming instruction.

In addition to the standard immediate and extended opcode fields, there are several other non-standard fields in the instruction set. Loads and stores have tags (LSIDs) to specify their program order to the memory system. Branches contain tags to assist the block-level branch predictor. Each instruction also contains a two-bit predicate field, which specifies whether the instruction should execute as soon as its operands arrive, or whether it should wait for a predicate operand to arrive before firing. In the latter case, the instruction fires only if the arriving predicate matches its predicate type (true or false). Having binary predicate values reduces the number of overhead instructions needed to invert predicates.

The predicate bits are used to implement predication in an EDGE ISA. Predication can convert control flow (branches) into data flow (predicate values) that guard instructions executed in case they belong to a non-taken path. Elimination of branches allows multiple EDGE instruction blocks to be combined into a single atomic instruction block, which exposes concurrency to software and reduces block management control overheads. Creating larger blocks improves the effectiveness and performance of EDGE architectures.



Instructions

Load and Store Instructions		
LB	Load Byte	L
LH	Load Halfword	L
LW	Load Word	L
LD	Load Doubleword	L
SB	Store Byte	S
SH	Store Halfword	S
SW	Store Word	S
SD	Store Doubleword	S
Integer Arithmetic Instructions		
ADD	Add	G
ADDI	Add Immediate	I
SUB	Subtract	G
SUBI	Subtract Immediate	I
MUL	Multiply	G
MULI	Multiply Immediate	I
DIVS	Divide Signed	G
DIVSI	Divide Signed Immediate	I
DIVU	Divide Unsigned	G
DIVUI	Divide Unsigned Immediate	I
Integer Logical Instructions		
AND	Bitwise AND	G
ANDI	Bitwise AND Immediate	I
OR	Bitwise OR	G
ORI	Bitwise OR Immediate	I
XOR	Bitwise XOR	G
XORI	Bitwise XOR Immediate	I

Integer Shift Instructions		
SLL	Shift Left Logical	G
SLLI	Shift Left Logical Immediate	I
SRL	Shift Right Logical	G
SRLI	Shift Right Logical Immediate	I
SRA	Shift Right Arithmetic	G
SRAI	Shift Right Arithmetic Immediate	I
Integer Extend Instructions		
EXTSB	Extend Signed Byte	G
EXTSH	Extend Signed Halfword	G
EXTSW	Extend Signed Word	G
EXTUB	Extend Unsigned Byte	G
EXTUH	Extend Unsigned Halfword	G
EXTUW	Extend Unsigned Word	G
Integer Test Instructions		
TEQ	Test EQ	G
TEQI	Test EQ Immediate	I
TLT	Test LT	G
TLTI	Test LT Immediate	I
TLE	Test LE	G
TLEI	Test LE Immediate	I
TLTU	Test LT Unsigned	G
TLTUI	Test LT Unsigned Immediate	I
TLEU	Test LE Unsigned	G
TLEUI	Test LE Unsigned Immediate	I

Floating-Point Arithmetic Instructions		
FADD	FP Add	G
FSUB	FP Substract	G
FMUL	FP Multiply	G
FDIV	FP Divide	G
Floating-Point Test Instructions		
FEQ	FP Test EQ	G
FLT	FP Test LT	G
FLE	FP Test LE	G
Floating-Point Conversion Instructions		
FITOD	Convert Integer to Double FP	G
FDTOI	Convert Double FP to Integer	G
FSTOD	Convert Single FP to Double FP	G
FDTOS	Convert Double FP to Single FP	G
Control Flow Instructions		
BR	Branch	B
BRO	Branch with Offset	B
CALL	Call	B
CALLO	Call with Offset	B
RET	Return	B
SCALL	System Call	B
Miscellaneous Instructions		
NULL	Nullify Output	G
MOV	Move	G
MOVI	Move Immediate	I
GENS	Generate Signed Constant	C
GENU	Generate Unsigned Constant	C
APP	Append Constant	C
NOP	No Operations	C

Figure 1: The TRIPS Instruction Set Architecture (ISA)

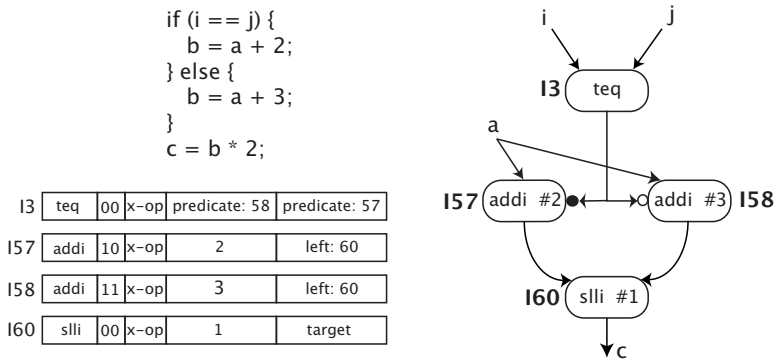


Figure 2: Predication in the TRIPS ISA

3.3.1 Predication

Figure 2 shows an example of a simple C-code fragment of an if-then-else and a diagram representing its if-converted dataflow graph. The right-most field in the instruction is the target identifier, which specifies the consumer of the instruction's result. The TRIPS ISA allows up to 128 instructions within each block.

The TRIPS ISA but must follow a number of rules to produce well-formed, predicated blocks:

1. Any instruction (except for a few specific data movement and constant generation instructions) may be predicated. A two-bit predicate field indicates whether an instruction is predicated and on what polarity of the arriving predicate the instruction should be executed.
2. For a predicated instruction to fire and execute, it must receive all of its data operands and a matching predicate operand. A matching predicate is one that matches the polarity of the waiting instruction. For example, an instruction waiting for a "false" predicate will only fire when a "false" predicate arrives.
3. Multiple instructions may target the predicate operand of an instruction, but at most one may deliver a matching predicate. This capability permits aggressive instruction merging.
4. To deliver a predicate to multiple predicated instructions, the compiler must construct the necessary fanout tree using a series of multi-target `move` instructions.
5. The predicated dataflow graphs must preserve the exception behavior of an unpredicated program, meaning that the same exceptions must be detected at the TRIPS block boundaries.

Dataflow predication provides the ability to compute compound predicates efficiently, while also exploiting early mispredication detection and implicit predication. The overheads of dataflow predication, as instantiated in TRIPS, require a two-bit field per instruction and fanout instructions for routing predicates to more than two predicate consumers. Previous architectures also had significant overheads for predication. Partially predicated dataflow ISAs added extra split and merge instructions. VLIW architectures required larger per-instruction fields (e.g. six bits) to specify predicate registers.

Figure 2 also illustrates predicate generation. The test instruction (`teq`) receives `i` and `j`, computes a predicate, and sends it to the two `addi` instructions. Note that the `addi` instructions are predicated on opposite polarities; the black circle indicates predication on true, while the white circle indicates predication on false. When the `addi` instructions receive both `a` and the predicate, the instruction with the matching predicate fires and delivers the result to the subsequent shift (`slli`) instruction. Since only one `addi` instruction fires, the shift will receive only one token representing the updated value of `b`.

Figure 2 also shows the encodings of the four instructions. The instruction fields include opcode (7 bits), predicate (2 bits), extended opcode (5 bits), immediate or target 2 (9 bits), and target 1 (9 bits). The predicate field specifies whether the instruction is predicated on a true predicate (PR = 11), predicated on a false predicate (PR = 10), or unpredicated (PR = 00). In this example, the unpredicated `teq` instruction, which has a PR field of 00, produces a "true" predicate, in which the low-order bit of the value routed to the consumers is equal to one. The `teq` has two targets, 57 and 58, which correspond to the predicate operands of the two `addi` instructions respectively. Each `addi` instruction has only one target as the second target field is needed to encode the immediate value.

3.4 Code Examples

Substantial refinements were made to the concept of EDGE architectures as part of the Phase 2 support. In addition to many of the predication features, the TRIPS ISA was defined in detail. In this section, we show four examples of how a simple routine is successively lowered into an EDGE ISA encoding.

Vector Add loop: We demonstrate EDGE-related transformations using a vector add loop example, since it is relatively simple to follow. Below we show a snippet of C code for the loop in which each vector is 1024 elements long.

```
for (i = 0; i < 1024; i++)
{
  C[i] += (A[i] + B[i]);
}
```

Unrolled Vector Add example: As discussed later in this report, the TRIPS compiler unrolls frequently iterated loops to fill out each instruction block. In this example, vagaries of the TRIPS EDGE ISA (i.e., the limit of no more than 32 loads and/or stores per block) limit the number of iterations from the above loop that may be packed into a single block. The loop body is replicated eight times, as shown below, expanded to fill a single instruction block as much as possible.

```
C[i]   += A[i]   + B[i]
C[i+1] += A[i+1] + B[i+1]
C[i+2] += A[i+2] + B[i+2]
C[i+3] += A[i+3] + B[i+3]
C[i+4] += A[i+4] + B[i+4]
C[i+5] += A[i+5] + B[i+5]
C[i+6] += A[i+6] + B[i+6]
C[i+7] += A[i+7] + B[i+7]
i+=8
test i < 1024
```

TIL Vector Add code: The below code represents the vector-add output of the compiler before assembly and instruction identifier assignment. We defined a RISC-like intermediate compiler form called the TRIPS Intermediate Language, or TIL. This form is readable by the compiler, and is significantly easier to manually tune than TRIPS Assembly Language (TASL). Since TIL is generated by the compiler after block formation, each block contains a `.bbegin` and a `.bend` identifier. There are a number of read instructions in each block that read global registers and identify which instructions in the block need those register values to commence execution. At the end of the block, there are multiple write registers to preserve values other than memory values. The code below has not yet been lowered into target format, meaning that instructions still specify abstract source and destination registers within the block. However, load/store IDs are specified in the instructions. Near the end of the block, note the two branch instructions, which are predicated on opposite predicate values, guaranteeing that only one of the two will execute when the block is executed.

```
.bbegin vadd$1
  read  $t0, $g70
  read  $t1, $g71
  read  $t2, $g72
  read  $t3, $g73
  ld    $t4, ($t1) L[0]
  ld    $t5, ($t2) L[1]
  ld    $t6, ($t3) L[2]
  fadd  $t7, $t5, $t6
  fadd  $t8, $t4, $t7
  sd    ($t1), $t8 S[3]
  ld    $t9, 8($t1) L[4]
  ld    $t10, 8($t2) L[5]
  ld    $t11, 8($t3) L[6]
  fadd  $t12, $t10, $t11
  fadd  $t13, $t9, $t12
  sd    8($t1), $t13 S[7]
  ld    $t14, 16($t1) L[8]
  ld    $t15, 16($t2) L[9]
  ld    $t16, 16($t3) L[10]
  fadd  $t17, $t15, $t16
  fadd  $t18, $t14, $t17
  sd    16($t1), $t18 S[11]
  ld    $t19, 24($t1) L[12]
  ld    $t20, 24($t2) L[13]
  ld    $t21, 24($t3) L[14]
  fadd  $t22, $t20, $t21
  fadd  $t23, $t19, $t22
  sd    24($t1), $t23 S[15]
  ld    $t24, 32($t1) L[16]
  ld    $t25, 32($t2) L[17]
  ld    $t26, 32($t3) L[18]
  fadd  $t27, $t25, $t26
  fadd  $t28, $t24, $t27
  sd    32($t1), $t28 S[19]
  ld    $t29, 40($t1) L[20]
  ld    $t30, 40($t2) L[21]
  ld    $t31, 40($t3) L[22]
  fadd  $t32, $t30, $t31
  fadd  $t33, $t29, $t32
  sd    40($t1), $t33 S[23]
  ld    $t34, 48($t1) L[24]
  ld    $t35, 48($t2) L[25]
  ld    $t36, 48($t3) L[26]
  fadd  $t37, $t35, $t36
  fadd  $t38, $t34, $t37
  sd    48($t1), $t38 S[27]
  ld    $t39, 56($t1) L[28]
  ld    $t40, 56($t2) L[29]
  ld    $t41, 56($t3) L[30]
  fadd  $t42, $t40, $t41
  fadd  $t43, $t39, $t42
  sd    56($t1), $t43 S[31]
  addi  $t45, $t0, 8
  addi  $t47, $t1, 64
  addi  $t49, $t2, 64
  addi  $t51, $t3, 64
  genu  $t52, 1024
  tlt   $t54, $t45, $t52
```

```
  bro_t<$t54> vadd$1
  bro_f<$t54> vadd$2
  write $g70, $t45
  write $g71, $t47
  write $g72, $t49
  write $g73, $t51
.bend
```

TASL Vector Add code: Finally, the below code example displays the same unrolled vector-add loop lowered into TASL code. There are three major differences between this code and the higher-representation TIL code. First, the code has been lowered into target format, meaning that each instruction specifies its operand target instructions, not its source operands. Second, each instruction has been assigned an identifier, shown on the left, which is used to identify the instructions. (These identifiers are implicit, and are actually determined based on an instruction's position within the block). Finally, the "mov" instructions shown near the top of the block represent values being sent to many consumers within the block. These instructions are not inserted into a block until after the blocks are lowered into TASL format.

```
.bbegin vadd$1
R[2]   read G[70] N[14,0]
R[3]   read G[71] N[43,0] N[3,0]
R[0]   read G[72] N[48,0] N[12,0]
R[1]   read G[73] N[18,0] N[13,0]

N[3]   mov N[7,0] N[9,0]
N[7]   mov N[11,0] N[2,0]
N[9]   mov N[4,0] N[37,0]
N[11]  mov N[15,0] N[6,0]
N[2]   mov N[1,0] N[75,0]
N[4]   mov3 N[32,0] N[78,0] N[64,0]
N[37]  mov3 N[49,0] N[65,0] N[84,0]
N[15]  mov3 N[19,0] N[109,0] N[10,0]
N[6]   mov3 N[108,0] N[5,0] N[106,0]
N[1]   mov3 N[0,0] N[107,0] N[33,0]
N[12]  mov N[16,0] N[76,0]
N[16]  mov N[44,0] N[21,0]
N[76]  mov3 N[66,0] N[77,0] N[110,0]
N[44]  mov3 N[72,0] N[96,0] N[73,0]
N[21]  mov N[35,0] N[42,0]
N[13]  mov N[17,0] N[41,0]
N[17]  mov N[8,0] N[20,0]
N[41]  mov3 N[69,0] N[97,0] N[34,0]
N[8]   mov3 N[36,0] N[68,0] N[40,0]
N[20]  mov N[46,0] N[50,0]
N[32]  ld L[0]  N[74,0]
N[66]  ld L[1]  N[70,0]
N[69]  ld L[2]  N[70,1]
N[70]  fadd N[74,1]
N[74]  fadd N[78,1]
N[78]  sd S[3]
N[64]  ld L[4] 8 N[45,0]
N[77]  ld L[5] 8 N[38,0]
N[97]  ld L[6] 8 N[38,1]
N[38]  fadd N[45,1]
N[45]  fadd N[49,1]
N[49]  sd S[7] 8
N[65]  ld L[8] 16 N[80,0]
N[110] ld L[9] 16 N[81,0]
```

```
N[34]  ld L[10] 16 N[81,1]
N[81]  fadd N[80,1]
N[80]  fadd N[84,1]
N[84]  sd S[11] 16
N[19]  ld L[12] 24 N[105,0]
N[72]  ld L[13] 24 N[101,0]
N[36]  ld L[14] 24 N[101,1]
N[101] fadd N[105,1]
N[105] fadd N[109,1]
N[109] sd S[15] 24
N[10]  ld L[16] 32 N[104,0]
N[96]  ld L[17] 32 N[100,0]
N[68]  ld L[18] 32 N[100,1]
N[100] fadd N[104,1]
N[104] fadd N[108,1]
N[108] sd S[19] 32
N[5]   ld L[20] 40 N[102,0]
N[73]  ld L[21] 40 N[98,0]
N[40]  ld L[22] 40 N[98,1]
N[98]  fadd N[102,1]
N[102] fadd N[106,1]
N[106] sd S[23] 40
N[0]   ld L[24] 48 N[103,0]
N[35]  ld L[25] 48 N[99,0]
N[46]  ld L[26] 48 N[99,1]
N[99]  fadd N[103,1]
N[103] fadd N[107,1]
N[107] sd S[27] 48
N[33]  ld L[28] 56 N[71,0]
N[42]  ld L[29] 56 N[67,0]
N[50]  ld L[30] 56 N[67,1]
N[67]  fadd N[71,1]
N[71]  fadd N[75,1]
N[75]  sd S[31] 56
N[14]  addi 8 N[22,0]
N[22]  mov N[79,0] W[2]
N[43]  addi 64 W[3]
N[48]  addi 64 W[0]
N[18]  addi 64 W[1]
N[39]  genu 1017 N[79,1]
N[79]  tlt N[83,p] N[111,p]
N[83]  bro_t vadd$1
N[111] bro_f vadd$2

W[2]   write G[70]
W[3]   write G[71]
W[0]   write G[72]
W[1]   write G[73]
.bend
```

3.5 Summary

EDGE ISAs, developed as a part of this DARPA contract, are a fundamental new class of instruction set (in addition to RISC, CISC, and VLIW instruction sets) that offer great potential for execution on future, highly distributed microarchitectures. The block-atomic execution permits low per-instruction control overheads, and efficient execution across a distributed substrate. The direct instruction communication within a block permits efficient dataflow execution,

without restricting the languages that can be compiled to an EDGE architecture. Outside of each instruction block, execution appears familiar, with communication through registers and memory. As discussed later in this report, compiling to an EDGE instruction set is quite feasible.

The refined instantiation of the TRIPS EDGE ISA, developed in Phase 1 and improved in Phase 2, still contains some overheads that hamper performance.

First, the number of move instructions was much larger than we expected and must be reduced significantly. These move instructions are needed to fan operands out to multiple consumers, as happens often with predicates and base addresses for memory accesses. Additionally, the large blocks with the large block headers consume too much memory, creating instruction cache pressure as well as binary bloat. We have taken steps to address both of these issues, and will discuss some of them in Section 10 of this report.

4 TRIPS Microarchitecture

To address the wire delay and complexity of a high-ILP (instruction-level parallel) processor, the TRIPS microarchitecture is explicitly partitioned into tiles that are connected via microarchitectural networks (or *micronets*) that route control and data among the tiles. Micronets provide high-bandwidth, flow-controlled transport for control and/or data in a wire-dominated processor by connecting the multiple tiles, which are clients on one or more micronets. Higher-level microarchitectural protocols direct global control across the micronets and tiles in a manner invisible to software.

In this section, we describe the tile partitioning, micronet connectivity, and distributed protocols that provide global services in the TRIPS processor, including distributed fetch, execution, flush, and commit [43, 7, 55]. Tiled architectures such as RAW [70] use static orchestration to manage global operations, but in a dynamically scheduled, distributed architecture such as TRIPS, hardware protocols are required to provide the necessary functionality across the processor.

To understand the design complexity, timing, area, and performance issues of this dynamic tiled approach, we implemented the TRIPS design in a 170M transistor, 130nm ASIC, the details of which are described in Section 5. This prototype chip contains two processor cores, each of which implements an EDGE instruction set architecture [7], is up to four-way multithreaded, and can execute a peak of 16 instructions per cycle. Each processor core contains 5 types of tiles communicating across seven micronets: one for data, one for instructions, and five for control used to orchestrate distributed execution. TRIPS prototype tiles range in size from 1-9mm^2. Four of the principal processor elements–instruction and data caches, register files, and execution units–are each subdivided into replicated copies of their respective tile type–for example, the instruction cache is composed of five instruction cache tiles, while the computation core is composed of 16 execution tiles.

The tiles are sized to be small enough so that wire delay within the tile is less than one cycle, and can largely be ignored from a global perspective. Each tile interacts only with its immediate neighbors through the various micronets, which have roles such as transmitting operands between instructions, distributing instructions from the instruction cache tiles to the execution tiles, or communicating control messages from the program sequencer. By avoiding any global wires or broadcast busses–other than the clock, reset tree, and interrupt signals–this design is inherently scalable to smaller processes, and is less vulnerable to wire delays than conventional designs.

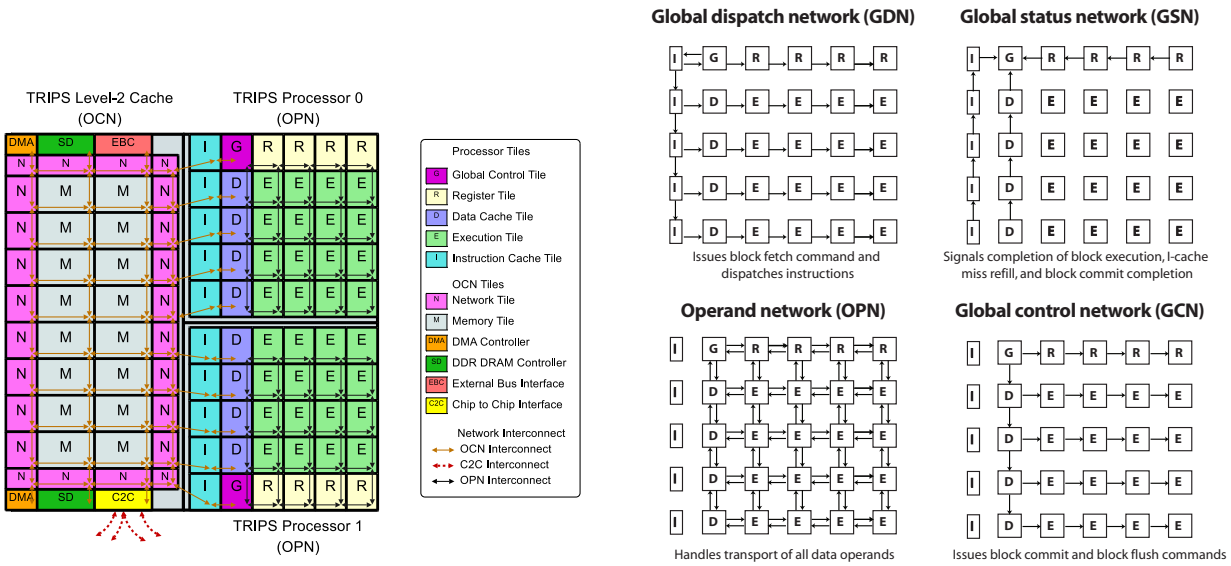
4.1 Processor Microarchitecture

The goal of the TRIPS microarchitecture is a processor that is scalable and distributed, meaning that it has no global wires, is built from a small set of reused components on routed networks, and can be extended to a wider-issue implementation without recompiling source code or changing the ISA. Figure 3(a) shows the tile-level block diagram of the TRIPS prototype. The three major components on the chip are two processors and the secondary memory system, each connected internally by one or more micronets.

Each of the processor cores is implemented using five unique tiles: one global control tile (GT), 16 execution tiles (ET), four register tiles (RT), four data tiles (DT), and five instruction tiles (IT). The main processor core micronet is the operand network (OPN), shown in Figure 3(b). It connects all of the tiles except for the ITs in a two-dimensional, wormhole-routed, 5x5 mesh topology. The OPN has separate control and data channels, and can deliver one 64-bit data operand per link per cycle. A control header packet is launched on separate wires one cycle in advance of the data payload packet to accelerate wakeup and select for bypassed operands that traverse the network.

Each processor core contains six other micronets, one for instruction dispatch–the global dispatch network (GDN)–and five for control: global control network (GCN), for committing and flushing blocks; global status network (GSN), for transmitting information about block completion; global refill network (GRN), for I-cache miss refills; data status network (DSN), for communicating store completion information; and external store network (ESN), for determining store completion in the L2 cache or memory. Links in each of these networks connect only nearest neighbor tiles and messages traverse one tile per cycle. Figure 3(b) shows the links for four of these networks.

This type of tiled microarchitecture is composable at design time, permitting different numbers and topologies of tiles in



(a) TRIPS Chip Block Diagram

(b) TRIPS Networks

Figure 3: TRIPS Chip Design

new implementations with only moderate changes to the tile logic, and no changes to the software model. The particular arrangement of tiles in the prototype produces a core with 16-wide out-of-order issue, 64KB of L1 instruction cache, 32KB of L1 data cache, and 4 SMT threads. The microarchitecture supports up to eight TRIPS blocks in flight simultaneously, seven of them speculative if a single thread is running, or two blocks per thread if four threads are running. The eight 128-instruction blocks provide an in-flight window of 1,024 instructions.

The two processors communicate through the secondary memory system, which is interconnected by the On-Chip Network (OCN). The OCN is a 4x10, wormhole-routed mesh network, with 16-byte data links and four virtual channels. This network is optimized for cache-line sized transfers, although other request sizes are supported for operations like loads and stores to uncacheable pages. The OCN acts as the transport fabric for all inter-processor, L2 cache, DRAM, I/O, and DMA traffic.

4.1.1 Global Control Tile (GT)

Figure 4(a) shows the contents of the GT, which include the blocks' program counters (PCs), the instruction cache tag arrays, the I-TLB, and the next-block predictor. The GT handles TRIPS block management, including prediction, fetch, dispatch, completion detection, flush (on mispredictions and interrupts), and commit. It also holds control registers that configure the processor into different speculation, execution, and threading modes. Thus the GT interacts with all of the control networks and the OPN, to provide access to the block PCs.

The GT also maintains the current status of all eight in-flight blocks. When one of the block slots is free, the GT accesses the block predictor, which takes three cycles, and emits the predicted address of the next target block. Each block may emit only one "exit" branch, even though it may contain several predicated branches. The block predictor uses a branch instruction's three-bit exit field to construct exit histories instead of using taken/not-taken bits. The predictor has two major parts: an exit predictor and a target predictor. The predictor uses exit histories to predict one of eight possible block exits, employing a tournament local/gshare predictor similar to the Alpha 21264 [31] with 9K, 16K, and 12K bits in the local, global, and tournament exit predictors, respectively. The predicted exit number is combined with the current block address to access the target predictor for the next-block address. The target predictor contains four major structures: a branch target buffer (20K bits), a call target buffer (6K bits), a return address stack (7K bits), and a branch type predictor (12K bits). The BTB predicts targets for branches, the CTB for calls, and the RAS for returns. The branch type predictor

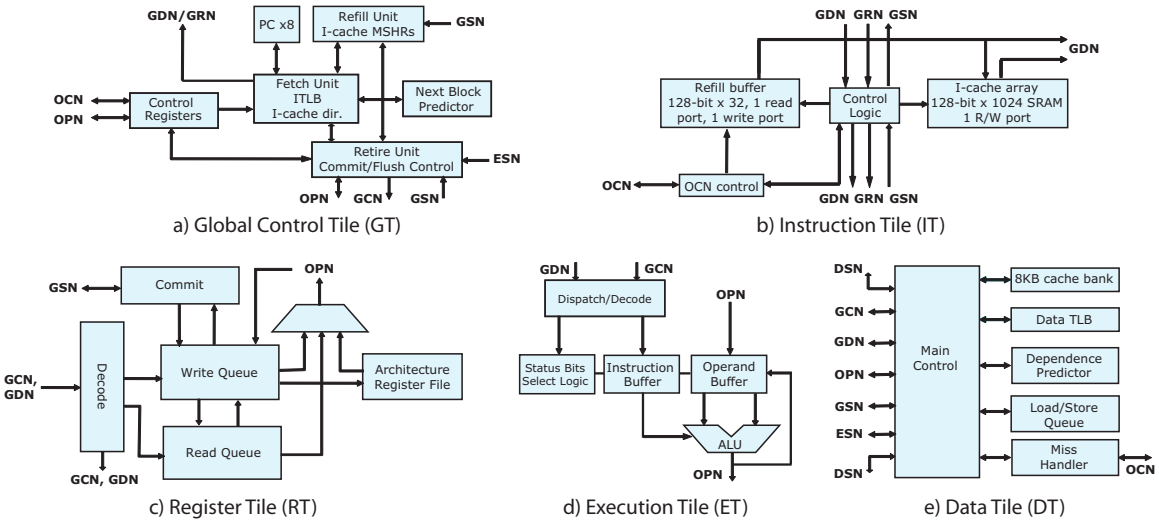


Figure 4: TRIPS Tile-Level Designs

selects among the different target predictions (call/return/branch/sequential branch). The distributed fetch protocol necessitates the type predictor; the predictor never sees the actual branch instructions, as they are sent directly from the ITs to the ETs.

4.1.2 Instruction Tile (IT)

Figure 4(b) shows an IT, which contains a 2-way, 16KB bank of the total L1 I-cache and acts as a slave to the GT, which holds the single tag array. Each of the five 16KB IT banks can hold a 128-byte chunk (for a total of 640 bytes for a maximum-sized block) for each of 128 distinct blocks.

4.1.3 Register Tile (RT)

To reduce power consumption and delay, the TRIPS microarchitecture partitions its many registers into banks, with one bank in each RT. The register tiles are clients on the OPN, allowing the compiler to place critical instructions that read and write from/to a given bank close to that bank. The compiler uses intra-block temporary values to capture the communication between many pairs of producing and consuming instructions, which reduces total register bandwidth requirements by approximately 70%, on average, compared to a RISC or CISC processor. The four distributed banks can thus provide sufficient register bandwidth with a small number of ports; in the TRIPS prototype, each RT bank has two read ports and one write port. Each of the four RTs contains one 32-register bank for each of the four SMT threads that the core supports, for a total of 128 registers per RT and 128 registers per thread across the RTs.

In addition to the four per-thread architecture register file banks, each RT contains a read queue and a write queue, as shown in Figure 4(c). These queues hold up to eight read and eight write instructions from the block header for each of the eight blocks in flight, and are used to forward register writes dynamically to subsequent blocks reading from those registers. The read and write queues perform a function equivalent to register renaming for a superscalar physical register file, but are less complex to implement due to the read and write instructions in the TRIPS ISA.

4.1.4 Execution Tile (ET)

As shown in Figure 4(d), each of the 16 ETs consists of a fairly standard single-issue pipeline, a bank of 64 reservation stations, an integer unit, and a floating-point unit. All units are fully pipelined except for the integer divide unit, which takes 24 cycles. The 64 reservation stations hold eight instructions for each of the eight in-flight TRIPS blocks. Each reservation station has fields for two 64-bit data operands and a one-bit predicate.

4.1.5 Data Tile (DT)

Figure 4(e) shows a block diagram of a single DT. Each DT is a client on the OPN and holds a single 2-way, 8KB L1 data cache bank, for a total of 32KB across the four DTs. Virtual addresses are interleaved across the DTs at the granularity of a 64-byte cache-line. In addition to the L1 cache bank, each DT contains a copy of the load/store queue (LSQ), a dependence predictor, a one-entry back-side coalescing write buffer, a data TLB, and a MSHR that supports up to 16 requests for up to four outstanding cache lines.

Because the DTs are distributed in the network, we implemented a memory-side dependence predictor, closely coupled with each data cache bank [56]. Although loads issue from the ETs, a dependence prediction occurs (in parallel) with the cache access only when the load arrives at the DT. The dependence predictor in each DT uses a 1024-entry bit vector. When an aggressively issued load causes a dependence misprediction (and subsequent pipeline flush), the dependence predictor sets a bit to which the load address hashes. Any load whose predictor entry contains a set bit is stalled until all prior stores have completed. Since there is no way to clear individual bit vector entries in this scheme, the hardware clears the dependence predictor after every 10,000 blocks of execution.

The hardest challenge in designing a distributed data cache was the memory disambiguation hardware. Since the TRIPS ISA restricts each block to 32 maximum issued loads and stores and eight blocks can be in flight at once, up to 256 memory operations may be in flight. However, the mapping of memory operations to DTs is unknown until their effective addresses are computed. Two resultant problems are: (a) determining how to distribute the LSQ among the DTs, and (b) determining when all earlier stores have completed–across all DTs–so that a held-back load can issue.

While neither centralizing the LSQ nor distributing the LSQ capacity across the four DTs were feasible options at the time, we solved the LSQ distribution problem largely by brute force. We replicated four copies of a 256-entry LSQ, one at each DT. This solution is wasteful and not scalable (since the maximum occupancy of all LSQs is 25%), but was the least complex alternative for the prototype. The LSQ can accept one load or store per cycle, forwarding data from earlier stores as necessary. Additional details on the DT design can be found in [56].

4.2 Distributed Microarchitectural Protocols

To enable concurrent, out-of-order execution on this distributed substrate, we implemented traditionally centralized microarchitectural functions, including fetch, execution, flush, and commit, with distributed protocols running across the control and data micronets.

4.2.1 Block Fetch Protocol

The fetch protocol retrieves a block of 128 TRIPS instructions from the ITs and distributes them into the array of ETs and RTs. In the GT, the block fetch pipeline takes a total of 13 cycles, including three cycles for prediction, one cycle for TLB and instruction cache tag access, and one cycle for hit/miss detection. On a cache hit, the GT sends eight pipelined indices out on the GDN to the ITs. Prediction and instruction cache tag lookup for the next block is overlapped with the fetch commands of the current block. Running at peak, the machine can issue fetch commands every cycle with no bubbles, beginning a new block fetch every eight cycles.

When an IT receives a block dispatch command from the GT, it accesses its I-cache bank based on the index in the GDN message. In each of the next eight cycles the IT sends four instructions on its outgoing GDN paths to its associated row of ETs and RTs. These instructions are written into the read and write queues of the RTs and the reservation stations in the ETs when they arrive at their respective tiles, and are available to execute as soon as they arrive. Since the fetch commands and fetched instructions are delivered in a pipelined fashion across the ITs, ETs, and RTs, the furthest ET receives its first instruction packet ten cycles and its last packet 17 cycles after the GT issues the first fetch command. While the latency appears high, the pipelining enables a high-fetch bandwidth of 16 instructions per cycle in steady state, one instruction per ET per cycle.

On an I-cache miss, the GT instigates a distributed I-cache refill, using the GRN to transmit the refill block's physical address to all of the ITs. Each IT processes the misses for its own chunk independently, and can simultaneously support one outstanding miss for each executing thread (up to four). When the two 64-byte cache lines for an IT's 128-byte block chunk return, and when the IT's south neighbor has finished its fill, the IT signals refill completion northward on the GSN. When the GT receives the refill completion signal from the top IT, it may issue a dispatch for that block to all ITs.

4.2.2 Distributed Execution

Dataflow execution of a block begins by the injection of block inputs by the RTs. An RT may begin to process an arriving read instruction even if the entire block has not yet been fetched. Each RT first searches the write queues of all older in-flight blocks. If no matching, in-flight write to that register is found, the RT simply reads that register from the architectural register file and forwards it to the consumers in the block via the OPN. If a matching write is found, the RT takes one of two actions. If the write instruction has received its value, the RT forwards that value to the read instruction's consumers. If the write instruction is still awaiting its value, the RT buffers the read instruction, which will be woken up by a tag broadcast when the pertinent write's value arrives.

Arriving OPN operands wake up instructions within the ET, which selects and executes enabled instructions. The ET uses the target fields of the selected instruction to determine where to send the resulting operand. Arithmetic operands traverse the OPN to other ETs, while load and store instructions' addresses and data are sent on the OPN to the DTs. Branch instructions deliver their next block addresses to the GT via the OPN.

An issuing instruction may target its own ET or a remote ET. If it targets its local ET, the dependent instruction can be woken up and executed in the next cycle using a local bypass path to permit back-to-back issue of dependent instructions. If the target is a remote ET, a control packet is formed the cycle before the operation will complete execution and sent to wake up the dependent instruction early. The OPN is tightly integrated with the wakeup and select logic. When a control packet arrives from the OPN, the targeted instruction is accessed and may be speculatively woken up. The instruction may begin execution in the following cycle as the OPN router injects the arriving operand directly into the ALU. Thus, for each OPN hop between dependent instructions, there will be one extra cycle before the consuming instruction executes.

Figure 5(a) shows an example of how a code sequence is executed on the RTs, ETs, and DTs. Block execution begins when the read instruction R[0] is issued to RT0, triggering delivery of R4 via the OPN to the left operand of two instructions, `teq` (N[1]) and `muli` (N[2]). When the test instruction receives the register value and the immediate "0" value from the `movi` instruction, it fires and produces a predicate which is routed to the predicate field of N[2]. Since N[2] is predicated on false, if the routed operand has a value of 0, the `muli` will fire, multiply the arriving left operand by four, and send the result to the address field of the `lw` (load word). If the load fires, it sends a request to the pertinent DT, which responds by routing the loaded data to N[33]. The DT uses the load/store IDs (0 for the load and 1 for the store, in this example) to ensure that they execute in the proper program order if they share the same address. The result of the load is fanned out by the `mov` instruction to the address and data fields of the store.

If the predicate's value is 1, N[2] will not inject a result into the OPN, thus suppressing execution of the dependent load. Instead, the `null` instruction fires, targeting the address and data fields of the `sw` (store word). Note that although two instructions are targeting each operand of the store, only one will fire, due to the predicate. When the store is sent to the pertinent DT and the block-ending call instruction is routed to the GT, the block has produced all of its outputs and is ready to commit. Note that if the store is nullified, it does not affect memory, but simply signals the DT that the store has

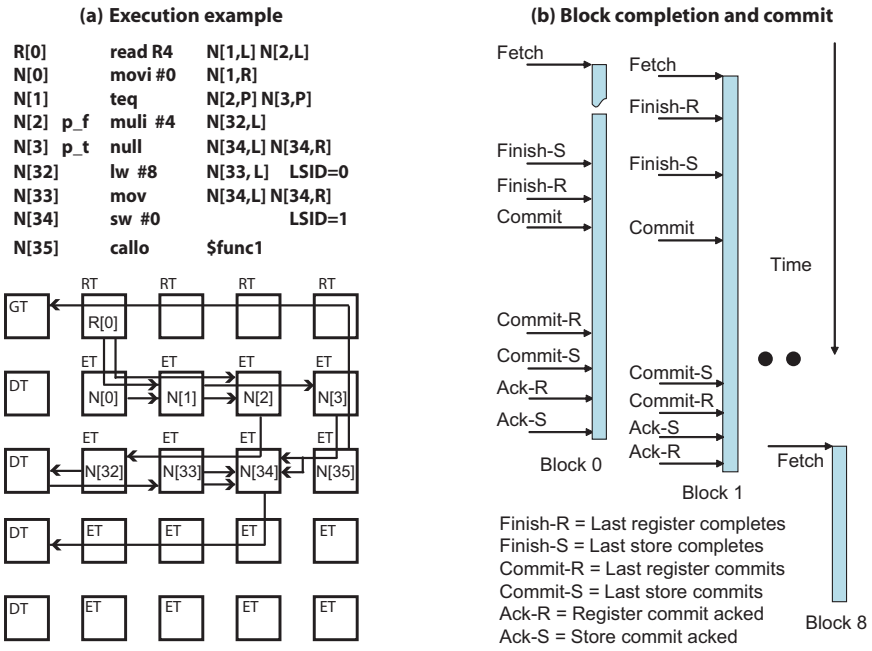


Figure 5: TRIPS Operational Protocols

issued. Nullified register writes and stores are used to ensure that the block always produces the same number of outputs for completion detection.

4.2.3 Block/Pipeline Flush Protocol

Because TRIPS executes blocks speculatively, a branch misprediction or a load/store ordering violation could cause periodic pipeline flushes. These flushes are implemented using a distributed protocol. The GT is first notified when a mis-speculation occurs, either by detecting a branch misprediction itself or via a GSN message from a DT indicating a memory-ordering violation. The GT then initiates a flush wave on the GCN which propagates to all of the ETs, DTs, and RTs. The GCN includes a block identifier mask indicating which block or blocks must be flushed. The processor must support multi-block flushing because all speculative blocks after the one that caused the mis-speculation must also be flushed. This wave propagates at one hop per cycle across the array nullifying unneeded instructions and OPN packets. As soon as it issues the flush command on the GCN, the GT may issue a new dispatch command to start a new block. Because both the GCN and GDN have predictable latencies, the instruction fetch/dispatch command can never catch up with or pass the flush command.

4.2.4 Block Commit Protocol

Block commit is the most complex of the microarchitectural protocols in TRIPS, since it involves the three phases illustrated in Figure 5(b): block completion, block commit, and commit acknowledgment. In phase one (block completion), a block is complete when it has produced all of its outputs, the number of which is determined at compile-time and consists of up to 32 register writes, up to 32 stores, and exactly one branch. After the RTs and DTs receive all of the register writes or stores for a given block, they inform the GT using the GSN. When an RT detects that all block writes have arrived, it informs its west neighbor. The RT completion message is daisy-chained westward across the RTs, until it reaches the GT indicating that all of the register writes for that block have been received.

Detecting store completion is more difficult since each DT cannot know a priori how many stores will be sent to it. To

enable the DTs to detect store completion, we implemented a DT-specific network called the DSN. Each block header contains a 32-bit *store mask*, which indicates the memory operations (encoded as an LSID bit mask) in the block that are stores. This store mask is sent to all DTs upon block dispatch. When an executed store arrives at a DT, its 5-bit LSID and block ID are sent to the other DTs on the DSN. Each DT then marks that store as received even though it does not know the store's address or data. Thus, a load at a DT learns when all previous stores have been received across all of the DTs. The nearest DT notifies the GT when all of the expected stores of a block have arrived. When the GT receives the GSN signal from the closest RT and DT, and has received one branch for the block from the OPN, the block is complete. Speculative execution may still be occurring within the block, down paths that will eventually be nullified by predicates, but such execution will not affect any block outputs.

During the second phase (block commit), the GT broadcasts a commit command on the Global Control Network and updates the block predictor. The commit command informs all RTs and DTs that they should commit their register writes and stores to architectural state. To prevent this distributed commit from becoming a bottleneck, we designed the logic to support pipelined commit commands. The GT can legally send a commit command on the GCN for a block when a commit command has been sent for all older in-flight blocks, even if the commit commands for the older blocks are still in flight. The pipelined commits are safe because each tile is guaranteed to receive and process them in order. The commit command on the GCN also flushes any speculative in-flight state in the ETs and DTs for that block.

The third phase (commit acknowledgement) acknowledges the completion of commit. When an RT or DT has finished committing its architectural state for a given block and has received a commit completion signal from its neighbor on the GSN (similar to block completion detection), it signals commit completion on the GSN. When the GT has received commit completion signals from both the RTs and DTs, it knows that the block is safe to deallocate, because all of the block's outputs have been written to architectural state. When the oldest block has acknowledged commit, the GT initiates a block fetch and dispatch sequence for that block slot.

4.3 Overheads of Distributed Microarchitecture

The physical design and implementation of the TRIPS chip were driven by the principles of partitioning and replication. The chip floorplan directly corresponds to the logical hierarchy of TRIPS tiles connected only by point-to-point, nearest-neighbor networks. The only exceptions to nearest-neighbor communication are the global reset and interrupt signals, which are latency tolerant and pipelined in multiple stages across the chip.

4.3.1 Area Overheads

The principal area overheads of the distributed design stem from the wires and logic needed to implement the tile-interconnection control and data networks shown in Table 1. The most expensive in terms of area are the two data networks: the operand network (OPN) and the on-chip network (OCN). In addition to the 141 physical wires per link, the OPN includes routers and buffering at 25 of the 30 processor tiles. The 4-port routers and the eight links per tile consume significant chip area and account for approximately 12% of the total processor area. Strictly speaking, this area is not entirely overhead as it takes the place of the bypass network, which would be much more expensive than the routed OPN for a 16-issue conventional processor. The OCN carries a larger area burden with buffering for four virtual channels at each of the 4-ported routers. It consumes a total of 14% of the total chip area, which is larger than a bus architecture for a smaller scale memory system, but necessary for the TRIPS NUCA cache. In general, the processor control networks themselves do not have a large area impact beyond the cost of the wires interconnecting the tiles. However, we found that full-chip routing was easily accomplished, even with the large number of wires. Across the entire chip, the area overhead associated with the distributed design stem largely from the on-chip data networks. The control protocol overheads are insignificant, with the exception of the load/store queue.

Table 1: TRIPS Control and Data Networks

Network	Use	Bits
Global Dispatch (GDN)	I-fetch	205
Global Status (GSN)	Block status	6
Global Control (GCN)	Commit/flush	13
Global Refill (GRN)	I-cache refill	36
Data Status (DSN)	Store completion	72
External Store (ESN)	L1 misses	10
Operand Network (OPN)	Operand routing	141 ($\times$8)
On-chip Network (OCN)	Memory traffic	138 ($\times$8)

4.3.2 Distributed Protocol Overheads

We examine the performance overheads of the distributed protocols via a simulation-based study using a cycle-level simulator, called tsim-proc, that models the hardware at a much more detailed level than higher-level simulators such as SimpleScalar. A performance validation effort showed that performance results from tsim-proc were on average within 4% of those obtained from the RTL-level simulator on our test suite and within 10% on randomly generated test programs. We use the methodology of Fields et al. [18] to attribute percentages of the critical path of the program to different microarchitectural activities and partitioning overheads.

The benchmark suite includes a set of microbenchmarks (dct8x8, sha, matrix, vadd), a set of kernels from a signal processing library (cfar, conv, ct, genalg, pm, qr, svd), a subset of the EEMBC suite (a2time01, bezier02, basefp01, rspeed01, tblook01), and a handful of SPEC benchmarks (mcf, parser, bzip2, twolf, and mgrid). In general, these are small programs or program fragments (no more than a few tens of millions of instructions) because we are limited by the speed of *tsim-proc*. The SPEC benchmarks use the reference input set, and we employ subsets of the program as recommended in [59]. These benchmarks reflect what can be run through our simulation environment, rather than benchmarks selected to leave an unrealistically rosy impression of performance. The TRIPS compiler toolchain takes C or Fortran77 code and produces complete TRIPS binaries that will run on the hardware. Although the TRIPS compiler is able to compile major benchmark suites correctly (i.e., EEMBC and SPEC2000) [60], there are many TRIPS-specific optimizations that are beyond the scope of the TRIPS contract. The current version of the compiler thus leaves some performance on the table because TRIPS blocks are small than desired.

While we report the results of the compiled code, we also employed some hand optimization on the microbenchmarks, kernels, and EEMBC programs in order to determine the potential of the architecture. We optimized compiler-generated TRIPS high-level assembly code by hand, feeding the result back into the compiler to assign instructions to ALUs and produce an optimized binary. Where possible, we report the results of the TRIPS compiler and the hand-optimized code.

To measure the contributions of the different microarchitectural protocols, we computed the critical path of the program and attributed each cycle to one of a number of categories. These categories include instruction distribution delays, operand network latency (including both hops and contention), execution overhead of instructions to fan operands out to multiple target instructions, time spent waiting for the GT to be notified that all of the block outputs (branches, registers, stores) have been produced, and the latency for the block commit protocol to complete. Table 2 shows the overheads as a percentage of the critical path of the program, and the column labeled “Other” includes components of the critical path also found in conventional monolithic cores including ALU execution time, and instruction and data cache misses.

The largest overhead contributor to the critical path is the operand routing, with hop latencies accounting for up to 34% and contention accounting for up to 25%. These overheads are a necessary evil for architectures with distributed execution units, although they can be mitigated through better scheduling to minimize the distance between producers and consumers along the critical path and by increasing the bandwidth of the operand network. For some of the benchmarks, the overheads of replicating and fanning out operand values can be as much as 12%. Most of the rest of the distributed protocol overheads are small, typically summing to less than 10% of the critical path. These results suggest that the overheads of the control networks are largely overlapped with useful instruction execution, but that the data networks

Table 2: Network Overheads and Preliminary Performance of Prototype

	Distributed network overheads as a percentage of program critical path						
Benchmark	IFetch	OPN Hops	OPN Cont.	Fanout Ops	Block Complete	Block Commit	Other
dct8x8	5.39	30.57	10.04	3.76	3.24	2.11	44.89
matrix	7.99	20.25	17.24	4.89	4.10	3.17	42.36
sha	0.57	17.91	6.29	11.73	0.10	0.66	62.74
vadd	7.41	17.66	13.79	5.61	5.99	7.48	42.06
cfar	3.75	32.06	9.39	9.78	2.44	0.99	41.59
conv	4.10	34.29	16.16	2.71	2.49	2.48	37.77
ct	6.23	18.81	16.25	6.04	3.65	3.79	45.23
genalg	3.85	18.60	5.76	8.82	2.21	0.62	60.14
pm	2.89	25.86	6.21	3.86	1.86	1.03	58.29
qr	4.53	22.25	8.85	11.97	2.72	2.23	47.45
svd	5.13	15.77	3.84	4.59	3.15	1.46	66.06
a2time01	4.94	13.57	6.47	9.52	2.05	4.02	59.43
bezier02	2.59	16.92	5.22	12.54	0.21	2.63	59.89
basefp01	3.36	13.63	5.44	6.34	2.74	2.90	65.59
rspeed01	0.76	28.67	10.61	11.77	0.39	0.14	47.66
tblook01	2.88	28.83	9.38	5.68	1.73	0.69	50.81
181.mcf	1.64	28.52	6.10	0.00	0.08	0.18	63.48
197.parser	2.96	30.76	3.99	0.84	0.30	0.66	60.49
256.bzip2	1.97	33.87	15.17	0.18	0.01	0.18	48.62
300.twolf	3.01	18.05	3.08	0.75	0.25	0.84	74.02
172.mgrid	5.06	18.61	25.46	4.03	3.00	2.78	41.06

could benefit from further optimization.

4.4 Non-Uniform Level-2 Cache System

Like each TRIPS core, the TRIPS on-chip memory system is tiled and distributed. The TRIPS prototype supports a 1MB static NUCA [32] array, organized into 16 memory tiles (MTs), each one of which holds a four-way, 64KB bank. An on-chip network interconnects the MTs with six controllers that connects the chip to the external interfaces. The two 133/266MHz DDR SDRAM controllers (SDC) each connect to an individual 1GB SDRAM DIMM. The chip-to-chip controller (C2C) extends the on-chip network to a four-port mesh router that gluelessly connects to other TRIPS chips. The two direct memory access (DMA) controllers can be programmed to transfer data to and from any two regions of the physical address space including addresses mapped to other TRIPS processors. Finally, the external bus controller (EBC) is the interface to a board-level PowerPC control processor. To reduce design complexity, we off-loaded much of the operating system and runtime control to this PowerPC processor.

In the TRIPS prototype chip, we implemented a static NUCA cache (S-NUCA-2) composed of 16 64-kbyte banks for a total of one megabyte. While dynamic NUCA caches can provide greater performance, we chose a simple design for the purpose of the prototyping effort. As shown in Figure 3(a), the NUCA banks are arranged in a 2-by-8 array to align with the multiple data cache ports of the two on-chip TRIPS processors. Each MT is identical in design an contains the bank storage along with the control logic and the network router. The NTs serve as interfaces into the NUCA array and include routers as well as a network address mapping table used for reconfiguring the memory system.

Memory addresses are interleaved on a cache-line basis across the banks with consecutive cache lines usually mapping to neighbor banks in the vertical dimension. We chose to make both the L1 and L2 cache lines sizes to be 64 bytes so that the interleaving pattern would be the same across the two levels of cache. The on-chip network (OCN) embedded into the NUCA array provides access from any L1 data cache bank to any NUCA bank. Each channel in the OCN is 16 bytes,

resulting in a four-flit message (plus one header flit) to transfer each 64 byte cache line. Thus five cycles are required to inject a complete cache line into the network. Each TRIPS processor's data cache bank is adjacent to a row of the NUCA array, enabling fast parallel access from the processors to the secondary memory system. NUCA bandwidth is highest and latency is lowest when the memory access is aligned across the physical row containing adjacent L1 and L2 cache banks. Bandwidth and latency degrade when requests and responses traverse to far reaches of the NUCA array.

4.4.1 TRIPS Physical Address Space

The TRIPS system employs 40-bit physical addresses for a total address space of one terabyte. This space is partitioned into four 256 gigabyte quadrants:

- Cached SDRAM: allows access to system memory via the L2 NUCA caches.
- Uncached SDRAM: allows direct access to system memory without going through the L2 caches.
- SRF: allows access to the on-chip NUCA banks configured as physically addressed memory. SRF stands for streaming register file which is one method of using the on-chip banks.
- Configuration: allows access to memory mapped configuration registers.

Each quadrant is further divided into 4GB regions, with one region from each quadrant assigned to each TRIPS chip (up to 64) in the system. There are a total of 256 regions (four per chip) within the TRIPS address space. The two most significant bits of the address (bits 39:38) determine the quadrant. The next six bits (bits 37:32) determine the region within each quadrant. The remaining 32 bits (31:0) provide an offset within the region.

The TRIPS system provides a global physical address space to all memory in the system. An access to remote memory from one processor is routed to the processor owning that remote memory via the TRIPS OCN and chip-to-chip (C2C) networks. To simplify the design, TRIPS has limited support for multiprocessor caching. Remote data may be cached in a local processor's L1 cache, but not in the local NUCA L2 cache. Accesses to remote data first access the remote L2 cache and may leave a cached copy of the data in the remote L2. TRIPS provides no hardware support for cache coherence.

4.4.2 Memory Tile Design

Each TRIPS memory tile (MT) consists of four 16-kilobyte data arrays and four 2.3-kilobyte tag arrays for a total of 64 kilobytes of four-way set associative data storage. Each MT also includes a 4-port on-chip network router, router interface logic, cache control logic, and one miss-status handling register (MSHR) to track outstanding cache misses instigated from the MT. All 16 MTs can form a one megabyte interleaved level-2 cache that can handle 16 simultaneous cache references as well as track 16 outstanding cache misses. The on-chip memory system allows for many more in-flight cache accesses because each MT and the on-chip network are pipelined. Each MT is approximately 6.5mm^2 of which the data and tag arrays occupy 2.2mm^2, the OCN router occupies 0.9mm^2, and the cache control logic occupies 3.4mm^2.

The MT provides aligned access to different data sizes, ranging from one to 64 bytes (in powers of two), to support the TRIPS cached and uncached load/store instructions of different sizes. In addition, the MT implements a single-byte swap access for the TRIPS atomic SWAP instruction. A cache hit in the MT takes three pipelined cycles from an arriving address to the departure of the first portion of the cache line. In steady state in which all accesses hit, an MT can service a new cache access every five cycles: one cycle to inject the message header for the MT response and four cycles for each of the 16-byte cache line portions. Cache misses are logged in the MSHR of the MT and directed to one of the two on-chip DRAM controllers via the OCN. Each MT includes five performance monitors to count the number of MT read and write accesses, the number of misses, the number of spills, and the number of demand fills. The performance monitors are accessed through memory-mapped load instructions that traverse the OCN like regular memory instructions.

Table 3: TRIPS NUCA Memory Latencies in Cycles

Memory Access	Minimum	Maximum
Local NUCA hit	7	21
Local NUCA miss	78	84
Remote NUCA hit	29	171
Remote NUCA miss	100	220

Table 3 shows the range of latencies (after leaving the processor core) for different types of memory accesses. Local accesses are to addresses mapped to the local processor's memory, while Remote accesses traverse the chip-to-chip network to an adjacent TRIPS chip. For each access type, the table shows the minimum and maximum latencies in cycles, which is determined largely by the number of hops traversed in the NUCA array. For the misses and the remote accesses, latency is also determined by the distance in hops from an MT to the DRAM and chip-to-chip controllers, respectively. All of the latencies in the table assume no network or bank contention. Contention can contribute significantly to memory access times. While worst-case latencies can be large, our experience shows that typical latencies are closer to the minimum latencies due to the memory parallelism and fast access to nearby banks enabled by NUCA caches.

4.4.3 Configuration

Because of its partitioned nature, the TRIPS NUCA cache can easily be configured for different cache sizes and mappings. The basic mechanisms for configuration are disabling the tag access within a memory tile, adjusting which bits of the address are mapped to index and tag, and controlling the routing of addresses to the MTs. Each MT has configuration registers and control logic that determine whether to use the cache tags and which bits to use as index and tag. In addition, the NUCA array is ringed by NTs which contain OCN routers and a configurable mapping table that determines how to map the L2 cached address space to the MTs. By adjusting the contents of the mapping table, system software can include or omit MTs from participating in the L2 cache. This flexibility enables three basic on-chip memory configurations.

- Shared NUCA cache: addresses are interleaved across all 16 MTs in the NUCA array on a cache line basis, creating a 1MB NUCA cache shared by both on-chip processors.
- Split NUCA cache: low physical addresses are interleaved on a cache line basis across the top eight MTs while the high physical addresses are interleaved in the same fashion across the bottom eight MTs. By controlling page allocation, the split mode can behave as two independent 512KB NUCA caches, one for each of the on-chip processors.
- Streaming Register File: the right eight MTs are configured as physical memory banks while the L2 cached address space is interleaved across the left eight MTs. This configuration produces a 512KB shared cache along with 512 KB of on-chip physical memory that can be used as a software controlled cache or scratchpad memory (sometimes called a streaming register file). The on-chip DMA controllers can overlap computation with the transfer of data between memory and the scratchpad.

While these are three preferred configurations of the TRIPS distributed on-chip memory system, many others are possible by altering which banks participate in the NUCA cache and address interleaving across the banks. Overall the TRIPS NUCA cache is somewhat less dense than a monolithic cache, due to the additional area required for the synthesized logic including the routers. However, the routed network is necessary for non-uniform access and has the added benefit of providing easy memory system configuration. While we chose bank size to be 64 KB so that the TRIPS system would have enough banks to be experimentally interesting, the ideal number of banks will depend on application characteristics and technology parameters.

4.5 Summary

When the first TRIPS paper appeared in 2001 [43], the high-level results seemed promising, but it was unclear (even to us) whether this technology was implementable in practice, or whether it would deliver the performance indicated by the high-level study. The TRIPS microarchitecture is an existence proof that the design challenges unanswered in 2001 were solvable; the distributed protocols we designed to implement the basic microarchitecture functions of instruction fetch, operand delivery, and commit are feasible and do not incur prohibitive overheads. The distributed control overheads are largely overlapped with instruction execution, the logic required to implement the protocols is not significant, and the pipelined protocols are not on critical timing paths. The distributed protocols have enabled us to construct a 16-wide, 1024-instruction window, out-of-order processor with large partitioned on-chip memories.

The tiled microarchitecture is amenable to polymorphism by retargetting the use of individual tiles to match the demands of an application. In TRIPS, reconfigurable hardware support is provided for (1) on-chip memory banks to be used for hardware or software controlled caches, (2) repartitioning of reservation stations to support single or multithreaded execution modes, and (3) programmable DMA controllers that can support both threaded and streaming models of parallelism. Our paper [54] discusses mechanisms for mapping different types of parallelism to the TRIPS substrate, while Section 10 details architectural enhancements that allow both static and dynamic processor composition to support polymorphism.

5 TRIPS Prototype System

The TRIPS prototype system is composed of custom-designed integrated circuit chips and custom-designed printed circuit boards. The chips were fabricated by IBM Microelectronics and the boards were designed by the USC Information Sciences Institute and fabricated by Bema Electronics, Inc. This section describes the chips, board, and hardware systems that resulted from the TRIPS project. Sections 6– 8 detail the software including the compiler, software development kit, and libraries.

5.1 TRIPS Chip Implementation

The TRIPS chip is implemented in the IBM CU-11 ASIC process, which has a drawn feature size of 130nm and 7 layers of metal. The chip itself includes more than 170 million transistors in a chip area of 18.30mm by 18.37mm, which is placed in a 47.5mm square ball-grid array package. The TRIPS chip design team included faculty, staff, and graduate students at UT-Austin and an IBM Microelectronics ASIC design team located in Austin, TX. UT-Austin was responsible for all architecture, logic design, verification, and timing. IBM supplied the physical design methodology and libraries, and was responsible for the physical design tasks including test infrastructure insertion, the final physical floorplan, placing and routing of all cells, and the tapeout process. Figure 6 shows a photograph of a TRIPS chip package and lists the final chip specification.

Controllers: In addition to the core tiles, the TRIPS chip also includes six controllers that are attached to the rest of the system via the on-chip network (OCN). The two 133/266MHz DDR SDRAM controllers (SDC) each connect to an individual 1GB SDRAM DIMM. The chip-to-chip controller (C2C) extends the on-chip network to a four-port mesh router that gluelessly connects to other TRIPS chips. These links nominally run at one-half the core processor clock and up to 266MHz. Each TRIPS prototype board includes four TRIPS chips and ports to extend the system to up to 32 TRIPS chips on eight boards. The two direct memory access (DMA) controllers can be programmed to transfer data to and from any two regions of the physical address space including addresses mapped to other TRIPS processors; the global physical address map contains memory regions for each processor in the system.

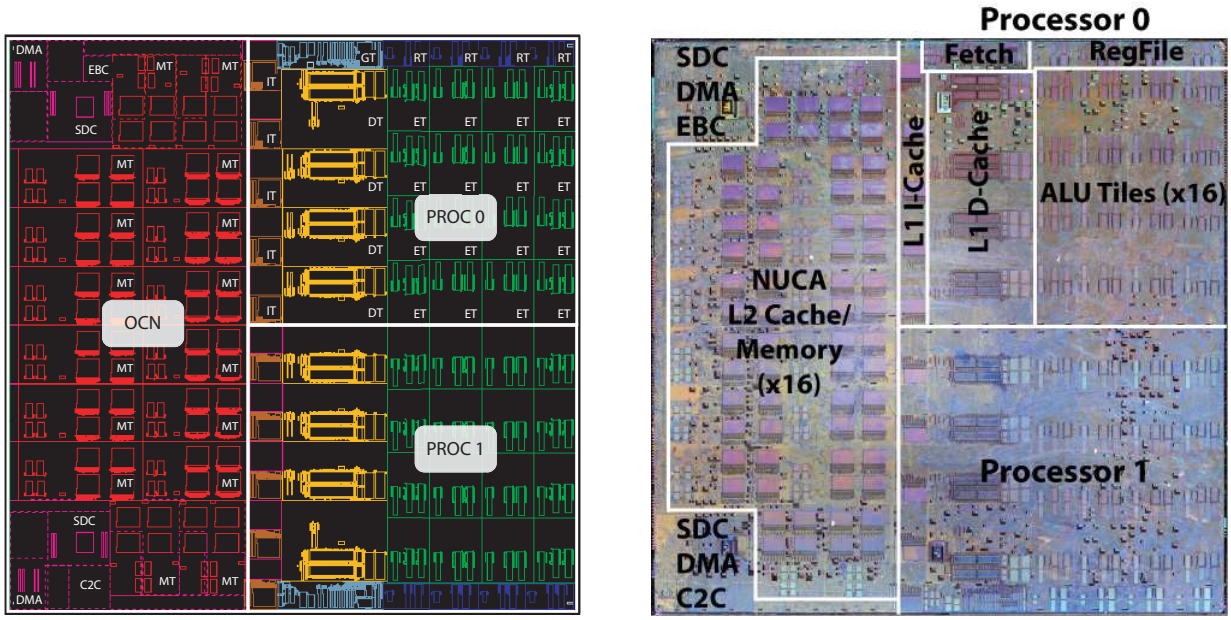
Finally, the external bus controller (EBC) is the interface to an on-board PowerPC control processor. To reduce design complexity, we chose to off-load much of the operating system and runtime control to this PowerPC processor. The EBC allows the PowerPC to read and write all TRIPS chip architectural state (memory, registers, etc.) and relays interrupt requests from TRIPS processors and DMA controllers to the PowerPC, which proxies system calls for the TRIPS chips on the board.

Process Technology	130nm ASIC with 7 metal layers
Die size	18.3mm x 18.37mm (336mm^2)
Package	47mm x 47mm BGA
Pin Count	626 Signals, 352 Vdd, 348 GND
Number of placed cells	6.1 million
Power measured	36W at 366MHz
Clock period	2.7ns (actual) 4.5ns (worst case simulated)

(a) TRIPS Specifications

(b) TRIPS Package

Figure 6: TRIPS Prototype Chip Specifications



(a) TRIPS Floorplan

(b) TRIPS Die Photo

Figure 7: TRIPS Prototype Chip

IOs and Test: The TRIPS chip includes nearly 600 signal I/Os, including 108 for each SDRAM interface, 312 for the chip-to-chip controller (39 pins per channel $\times$ four directions $\times$ input/output per direction), and 69 pins for the EBC. Figure 7(a) shows an annotated floorplan diagram of the TRIPS chip taken directly from the design database as well as a coarse area breakdown by function. Not shown in Figure 7(a) are the individual I/O cells, which are placed near the periphery of the chip. Some ETs, MTs, and DTs are larger than others to accommodate the placement of these I/O cells.

Finally, the ASIC methodology requires LSSD scan support for manufacturing testing and JTAG I/O boundary scan. In addition, we and our IBM partners added a scan controller to enable the scan chains to be used for silicon debug in functional mode by allowing scan access to most of the internal state. The TRIPS chip also includes two phase-locked loops (PLLs) to generate the clocks for the four on-chip clock domains (main clock, C2C clock, and two clocks for the DDR SDRAM controller). These clocks are asynchronous to one another and we use synchronizers when crossing the main clock, C2C clock and SDRAM clock boundaries. The C2C interface to other TRIPS chips is clocked in a source-synchronous fashion and incoming C2C packets are synchronized into the local domain before being used.

5.1.1 Floorplan

Figure 7(a) shows the boundaries of the TRIPS tiles, as well as the placement of register and SRAM arrays within each tile. We did not label the NTs that surround the OCN since they are so small. Also, for ease of viewing, we have omitted the individual logic cells from this plot. Table 4 lists the area breakdown of the major components of the chip. Each instance of a tile was individually placed and routed because IO cells are distributed through the chip and create blockages at different locations in different tiles. As a result all the instances of a tile do not look identical in this floorplan diagram.

5.1.2 Physical Design

The TRIPS design flow relies extensively on tile-level partitioning as well as a modular ASIC design flow. As a part of their ASIC services, IBM provides register and SRAM array generators that we used heavily not only for registers and memory, but also for branch prediction tables, instruction queues, and reservation stations. Through a university license,

Table 4: TRIPS Chip Area Breakdown

Overall Chip Area		Processor Area	
29%	Processor 0	30%	Functional Units (ALUs)
29%	Processor 1	4%	Register Files and Queues
21%	Level-2 Cache	10%	Level-1 Caches (I and D)
14%	On-chip Network	13%	Instruction Queues
7%	Other (controllers, etc.)	13%	Load/Store Queues
		12%	Operand Network
		2%	Next block predictor
		16%	Other

Synopsys provided their DesignWare suite which included synthesizable integer units, floating-point units, queues, and FIFOs. The design-time advantages of the ASIC flow are offset by greater area and slower clock rates relative to a custom design. However, the advantages of tile-level partitioning would apply directly to a custom VLSI design of TRIPS. Figure 7(b) shows a die photo of the chip. The close mapping of the final die photo to the logical organization shows the modular nature was carried through to the physical design stage as well.

Table 5 shows additional details on the design of each TRIPS tile. The Cell Instance column shows the number of placeable cell instances (logic gates) in each tile, which provides a relative estimate of the logic complexity of the tile. A placeable instance is a pre-defined macro available in the IBM library provided, examples of which include simple 2-input AND gates to SRAMs and register files. Array Bits indicates the total number of bits found in dense register and SRAM arrays on a per-tile basis, while Size shows the area of a representative of each type of tile. Although the logic for every instance of a tile is identical, each tile was individually placed and routed because IO cells are distributed through the chip and create blockages at different locations in different tiles. The representative area shows the area of one instance for each tile type. Tile Instances shows the total number of copies of that tile across the entire chip, and % Chip Area indicates the fraction of the total chip area occupied by that type of tile.

As shown in Table 5, the DT is certainly the most complex of the tiles, due in large part to the demands of an out-of-order memory system rather than the distributed nature of the TRIPS processor. Its cell count and area is skewed somewhat by the CAM arrays for the maximum sized load/store queues which had to be implemented from discrete latches, because no suitable dense array structure was available. We saw the same phenomenon in OPN and OCN routers. The large cell counts in the ET are due largely to the computational units, such as the floating point units, which are synthesized to the standard cell library rather than implemented using a custom datapath.

5.1.3 Verification

The partitioned nature of the TRIPS chip facilitated a highly hierarchical verification strategy. Each of the 11 tile design teams created a sophisticated self-checking testbench for their tile that employed both directed and random tests to exercise as many of the corner cases as possible. The random tests varied both test inputs and the timing of responses to tile requests. To assess coverage, we augmented each tile design with event counters, and ensured that the counters were exercised, all lines of Verilog were hit, and that the internal state machines hit all of the pertinent states. The tile design approach also provided opportunity for concurrent development and verification of the tiles before putting the tiles together and verification of the processor core or the full chip.

We also spent four person-months on performance verification. Using a suite of microbenchmarks, with some randomly generated programs, we reduced the average error between the low-level performance simulator and the RTL simulator from 10% on average to 3%. This effort uncovered sixteen performance bugs, ten of which turned out to be worth the effort to fix. The three most significant ones were fixing the issue priority in the ET, reducing the flush penalty by one cycle, and reordering predictor operations to eliminate an occasional pipeline bubble before issuing a fetch.

Table 5: TRIPS Tile Specifications

Tile	Function	Cell Instances	Array Bits	Size (mm^2)	Tile Instances	% Chip Area
GT	Processor control	51,684	93K	3.1	2	1.8
RT	Register file	26,284	14K	1.2	8	2.9
IT	Instruction cache	5,449	135K	1.0	10	2.6
DT	L1 Data cache	119,106	89K	8.8	8	21.0
ET	Instruction execution	83,887	13K	2.9	32	28.0
MT	L2 Data cache	60,115	542K	6.5	16	30.7
NT	OCN NW interface and routing	23,467	–	1.0	24	7.1
SDC	DDR SDRAM controller	64,441	6K	5.8	2	3.4
DMA	DMA controller	30,365	4K	1.3	2	0.8
EBC	External bus controller	28,547	–	1.0	1	0.3
C2C	Chip-to-chip communication controller	47,714	–	2.2	1	0.7
	Totals (for entire chip)	5.8M	11.5M	334	106	100.0

5.1.4 Timing

In the TRIPS project we decided not to aggressively push for timing optimizations given the schedule. Our very first goal at the high level design stage was a 1ns clock period, which after reviewing IBM's ASIC library and deciding on the final manufacturing processing we scaled down to 2.5ns. After full functional verification, synthesis, and a small set of timing optimizations our final clock period was 3.2ns. The bulk of timing optimization was in the tile-tile communication network path in the processor and the local bypass path in the execution tile. After complete physical design the final clock period at worst case process parameters is 4.5ns, which accounts for pessimistic clock skew and wiring parasitics from the final layout. To first order, this corresponds to approximately 32 fanouts-of-4 (where one FO4 is the latency for a single inverter to drive four copies of itself). By comparison, leading edge custom microprocessors are in the range of 15-20 FO4 [1]. A custom design style coupled with a more experienced design team, some amount of re-pipelining and more time devoted to timing optimization would likely be able to drive the TRIPS architecture into that same regime. Adding a more aggressive process and less conservative gates than a standard ASIC process would make the TRIPS clock rate competitive with that of a high-end commercial microprocessor. The maximum operation frequency of the final silicon was 366 MHz corresponding to clock period of 2.73 ns.

5.2 TRIPS System Design

The TRIPS design is scalable to a large number of chips and the overall system is designed with this scalability in mind. A TRIPS system consists of multiple TRIPS motherboards connected through the chip-to-chip interconnect and a flat memory space. All TRIPS motherboards are connected to one host PC through an Ethernet router. Each motherboard contains a PowerPC 440 GP which acts as the master processor for that board and four TRIPS daughtercards. Each TRIPS daughtercard contains a TRIPS chip and 2GB of DRAM. The design, layout, and testing of the TRIPS boards was performed by the Information Sciences Institute (ISI-East).

The smallest system consists of a single TRIPS motherboard (four TRIPS chips, eight TRIPS processors), as shown in Figure 8(a). Larger systems may be built using multiple motherboards. The largest deployed system currently consists of five TRIPS motherboards (20 TRIPS chips, 40 TRIPS processors) connected in series. Figure 8(b) shows a medium-sized system that consists of four TRIPS motherboards (16 TRIPS chips, 32 TRIPS processors). The motherboards are directly connected via their chip-to-chip (C2C) links using micro-coaxial ribbon cables. Figure 9 shows a photograph of the fully populated rack-based TRIPS system.

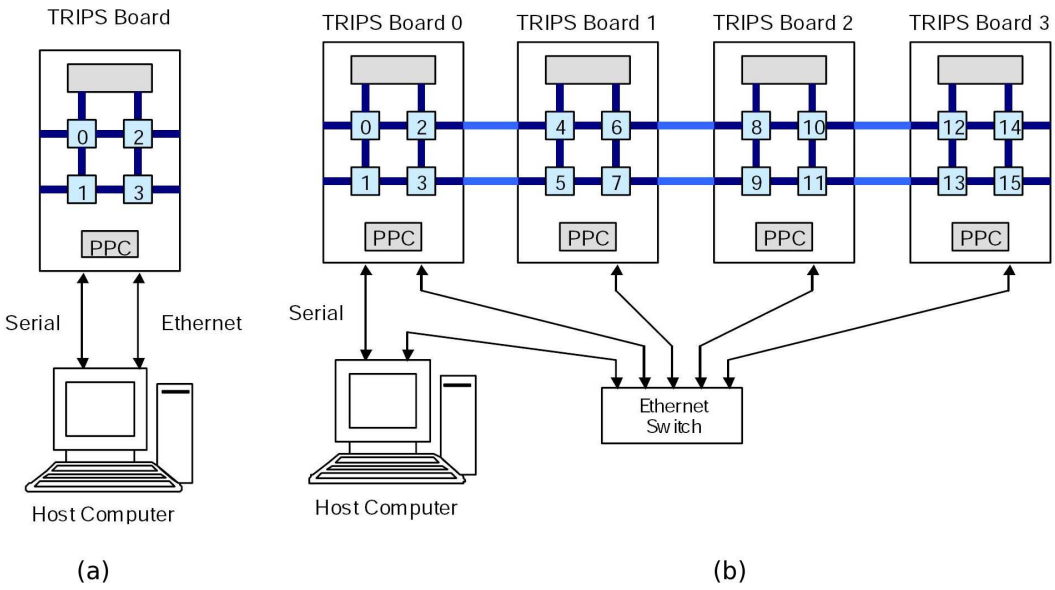
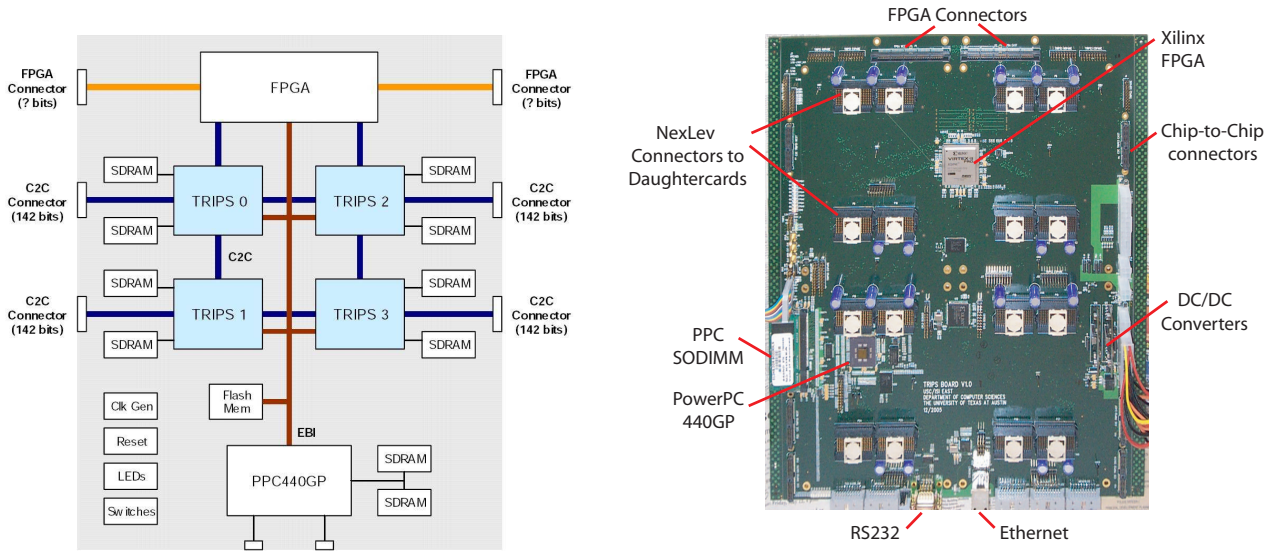


Figure 8: TRIPS System Design with (a) One Motherboard and (b) Four Motherboards

Figure 9: TRIPS Rack-based System with 5 Motherboards, 20 TRIPS Chips, and 40 TRIPS Processors



(a) Motherboard Floorplan

(b) Motherboard Photograph

Figure 10: TRIPS Motherboard

5.2.1 Motherboard

Figure 10 shows a block diagram and photograph of the a TRIPS motherboard. A motherboard measures approximately about 14” x 17” and when fully populated is almost completely covered by the 4 daughtercards. The components of the motherboard include:

- TRIPS daughtercard connectors: Each TRIPS motherboard has slots for four TRIPS daughtercards, each containing one TRIPS chip.
- TRIPS SDRAM: Each TRIPS chip interfaces to two DDR-266 SDRAM 184-pin Registered DIMMs (one DIMM slot per controller). Each DIMM slot can be populated with up to 1 GB of memory (for a total of 8 GB of TRIPS memory per board). Each slot is controlled by 2 chip selects and 13 address bits. (Each chip can support two additional 1 GB DIMMs, but we have elected to omit them for simplicity.) The SDRAM is also mounted on the daughtercard.
- FPGA Chip: Each TRIPS motherboard also includes an FPGA. This is intended to support system expansion and may allow the TRIPS board to interface to other types of boards. The FPGA is not required for normal operation.
- PPC440GP: The PowerPC 440GP chip is used as a master control processor, which configures and controls the TRIPS chips. It runs an embedded version of Linux and interacts with a host computer using a serial port and/or Ethernet connection. It is responsible for configuring the TRIPS chips, loading TRIPS programs into TRIPS memory, and providing runtime (kernel) services to TRIPS applications. The PPC440GP includes a large number of integrated components and I/O controllers.
- PPC SDRAM: The PPC440GP chip requires its own memory, implemented using a set of DDR-266 SDRAM 184-pin DIMMs. It supports up to 1 GB of memory (two 512 MB DIMMs).
- PPC Flash: Some amount of non-volatile (Flash) memory is available to store the OS image and a few local files (the PPC will also rely upon a network filesystem). The size is 64 MB. Part of the memory is useable for boot code.
- Clock Generator: A common reference clock is generated and distributed to every chip on the board(s). This reference clock runs at 33 MHz and is adjustable through switches.
- Reset Control: A master reset button allows all chips on a board to be reset.

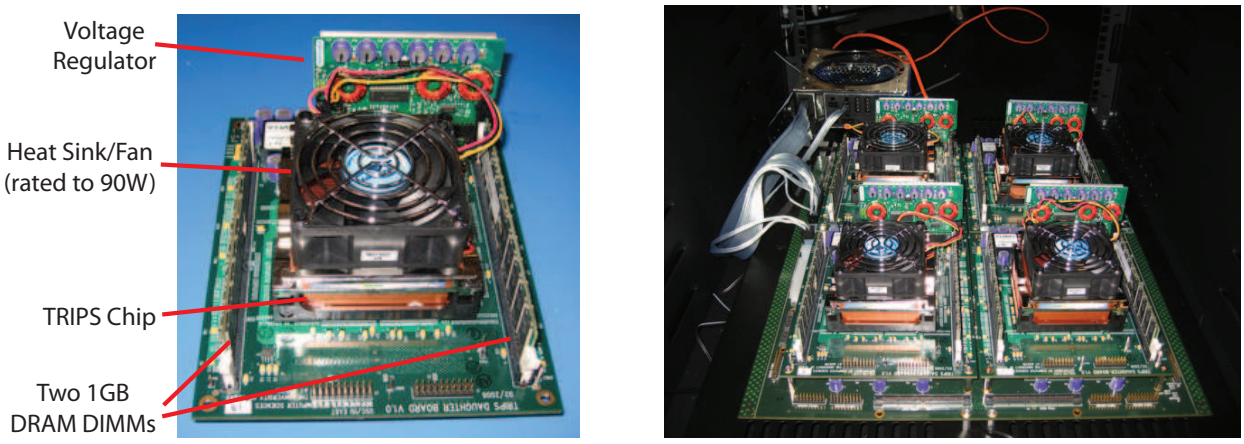
- LEDs: In addition to the normal power-on LEDs, some number of general-purpose LEDs are also present. These connect to each TRIPS chip (using GPIO) and are controllable using software.
- Switches: In addition to the normal power-on, reset, and clock configuration switches, some number of fixed-purpose and general-purpose switches are desired. These connect to each TRIPS chip (using GPIO) and are observable using software.
- System Bus (EBI): A simple asynchronous bus is used to connect the PPC440GP to various memory chips and to the TRIPS chips. This bus operates at up to 66 MHz and can read or write up to 32 bits of data at once. A separate chip select is assigned to each addressable chip.
- Chip-to-Chip (C2C) Network: A custom two-dimensional mesh network is used to connect multiple TRIPS chips within the system. Each chip is connected to its neighbors using 4 input links and 4 output links. Each link is 32 data bits wide (plus a few control signals), unidirectional, and expected to operate at up to 266 MHz. Additional C2C links connect the FPGA chip to the TRIPS chips.
- Interrupt Requests: A separate interrupt request line is routed from each TRIPS chip to the PPC440GP chip.
- Chip-to-Chip Connector:The C2C network is designed to extend across multiple boards. Dedicated connectors are needed to connect TRIPS chips on separate boards. These C2C links operate at a slower max frequency than the on-board C2C links.
- FPGA Connector: These connectors provide a back-side interface to each FPGA and should support a variety of system expansion scenarios.
- Ethernet Connector: The PPC440GP relies upon a 100-Mbps ethernet connection to communicate with a host computer.
- Serial Connector: The PPC440GP relies upon a standard serial port connector to communicate with a host computer.

5.2.2 Daughtercard

Because the TRIPS chip's package employs a fine-pitch ball-grid array, we were unable to seat the chips in sockets. Using daughtercards provides the capability to swap different TRIPS chips in and out of the system without requiring a full-motherboard replacement or electrical rework. This approach had the further advantage of allowing early board bringup and test (and software development) while the TRIPS chips were still being designed and fabricated.

The TRIPS daughtercard is far simpler than the motherboard, measuring at approximately 6" x 7.5". Figure 11(a) shows an annotated photograph of a daughtercard, displaying all of its components. Figure 11(b) shows a fully-populated motherboard with 4 daughtercards. Daughtercard components include:

- TRIPS Chip.
- Commodity heat sink and fan assembly, rated to 90W.
- DDR DIMM: Two DDR DIMM sockets for the TRIPS chip memory (1 GB per socket).
- NeXLev Connectors: Four C2C connectors that connect the C2C interfaces from the chip to the board.
- Commodity voltage regulator: power is brought onto the daughtercard from the PC ATX power supply at 12V and is stepped down to 1.5V, 2.5V, and 3.3V on-board for the different daughtercard components.



(a) Annotated Daughtercard Photo

(b) Populated TRIPS board

Figure 11: TRIPS Daughtercards

5.3 Summary

Several TRIPS systems of different sizes are being used for software development. Single board systems have been delivered to the Air Force Research Laboratory in Dayton, OH and to ISI-East in Arlington, VA. Several single-board systems as well as the rack-based system are deployed at The University of Texas at Austin.

6 TRIPS Compiler

EDGE architectures define a new ISA that places more responsibility on the compiler than RISC and CISC architectures. Compared to a compiler for a VLIW architecture, an EDGE compiler need not schedule when instructions execute relative to one another, but it must meet some additional constraints. An EDGE compiler and architecture contract contains two defining features: (1) block atomic execution, and (2) dataflow execution of the individual instructions that make up each block. The compiler must therefore encode the program as blocks of instructions and within a block encode the instructions in target dataflow form. Target dataflow form uses a specification of instruction placement. The compiler's scheduler selects an execution tile (E-tile) location for each dataflow instruction. Figure 12 shows the program divided in to blocks on the left side. Each block consists of dataflow instructions–the middle of the figure shows an example dataflow graph. The scheduler assigns instruction identifiers, that the hardware maps to E-tiles, as shown on the far right of the figure. An EDGE compiler must specify the units of block-atomic execution and within a block, where to execute instructions.

The TRIPS system delivered the first EDGE hardware implementation, and the first EDGE compiler for TRIPS hardware. The compiler technology demonstrated for the first time how to compile imperative programming languages (C and Fortran) to a dataflow architecture. Previous work on pure dataflow architectures required special purpose functional languages. By limiting dataflow execution to within a block, we were able to build hardware and a compiler that exploit the efficiencies of dataflow execution at a fine grain and that amortize coarser block grain overheads. Furthermore, this hybrid dataflow model requires no change to the programming model, while exploiting fine-grain concurrency within a block and coarser-grain concurrency between blocks.

The key challenges for the TRIPS compiler are (1) how to correctly generate code for EDGE architectures while obeying the TRIPS architectural resource constraints, such as fixed sized blocks and banked registers, and (2) how to form blocks to expose concurrency to the architecture for performance. The key to meeting these challenges was to simultaneously satisfy multiple dependent resource constraints, such as limits on block size and block composition, which required a new iterative algorithms to resolve competing resource constraints.

The new TRIPS compiler technology that we introduced includes (1) a new compiler structure that iteratively resolves competing resource constraints; (2) resolution of resource constraints to achieve correct code; (3) iterative block formation for high performance; (4) banked register allocation; and (5) instruction scheduling.

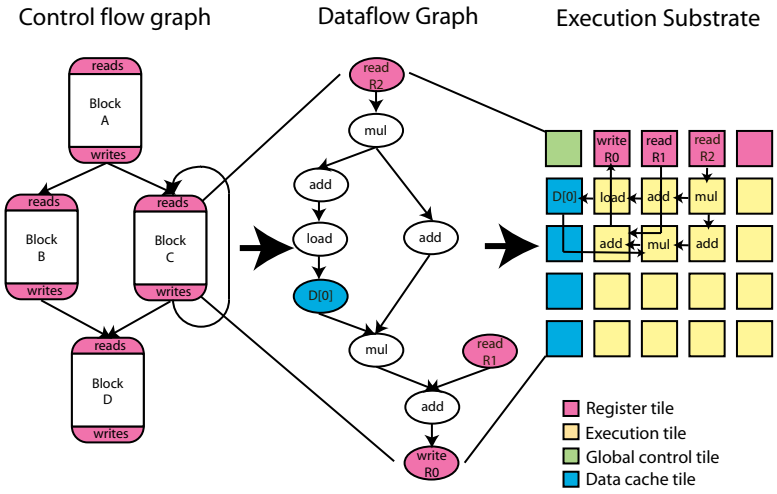


Figure 12: Block control flow graph and mapping of instructions on to a $4\times4\times8$ TRIPS Processor

6.1 Compiler Structure

This section gives an overview of the phases and intermediate forms used by the TRIPS compiler. Prior publications describe TRIPS block formation [37], scheduling [8], and satisfying block constraints [60]. Here, we describe the current version of the compiler which contains substantial differences from Smith et al. [60], and enhancements over Maher et al. [37].

Figures 13 and 14 illustrate the basic structure of the TRIPS compiler. The entire compiler is written in Java and includes over 700 classes. The frontend of the compiler accepts C, Fortran77, and a subset of Fortran90 as input. The front end applies scalar optimizations using Static Single Assignment form (SSA) as its intermediate representation. The frontend can generate Alpha assembly, SPARC assembly, C code (for testing purposes), and TIL (TRIPS intermediate language) [61].

Because programmers are familiar with RISC-assembly, we defined a RISC-like intermediate format, called TRIPS Intermediate Language [61]. It employs a simple and consistent syntax: *opcode result, operand1 [, operand2]*. The backend progressively lowers the TIL to meet the TRIPS block constraints. In the last phase of compilation, the TRIPS scheduler selects a location for each instruction and then encodes the positional information into each instruction and outputs TASL (TRIPS Assembly Language). TASL uses a target form which specifies the location of each instruction and the location(s) to send the instruction's result. Figure 15 shows an example code snippet in both TIL and TASL. Assembly language programmers can use either format.

6.2 Frontend Scalar Optimizations

The compiler frontend first performs inlining and loop optimizations such as unrolling and peeling. It then performs alias analysis and puts the program in Static Single Assignment form (SSA). It performs a variety of scalar optimizations on the SSA form, as listed in Figure 13. Each of these scalar phases leaves the program in SSA form, and thus all the phases may be reordered or applied repeatedly. The order and repetition of phases may be specified from the command line. In addition, command line options specify unrolling amounts, inlining amounts, and other optimization heuristic tuning parameters. For the SPARC and Alpha backends, we found our linear scan register allocator did better if many optimizations, e.g., loop invariant code, limited the number of temporaries they created. On TRIPS, we found that these heuristics did not perform as well as doing all legal scalar optimizations, without regard to temporary variable expansion, because same block temporaries need never go in registers. We therefore apply all these scalar optimizations every time they are applicable.

The only TRIPS specific scalar optimization in the frontend of the compiler is tree height reduction [27] which represents expressions as trees and then rewrites associative operations to minimize the tree height, thus maximizing instruction level parallelism. The rewriter also folds constants, placing the resulting expression in a canonical form. This algorithm produces short fat trees that use more registers and maximize instruction level parallelism. Because arithmetic expressions will almost always be evaluated within a TRIPS block, the temporary values they create will not require hardware registers. Instead, these temporaries will be encoded as direct instruction communication within a block.

The compiler can be configured to generate instrumented binaries which collect path, edge, or block profiles. Profiles are available to both the frontend and backend. Inlining uses profiles to guide the heuristics. Optimizations such as block formation also can take advantage of the profiles.

6.3 TRIPS Intermediate Language Form

The frontend produces TIL, which is a RISC-like 3-address intermediate representation [61]. At this point in compilation, the TIL does not reflect the block structure of EDGE ISAs or target form. The backend systematically transforms the TIL to more closely reflect the EDGE ISA until the TIL completely specifies correct blocks, including load/store identifiers and register assignment, but the individual instructions within the block are still in 3-address form. The TIL intermediate

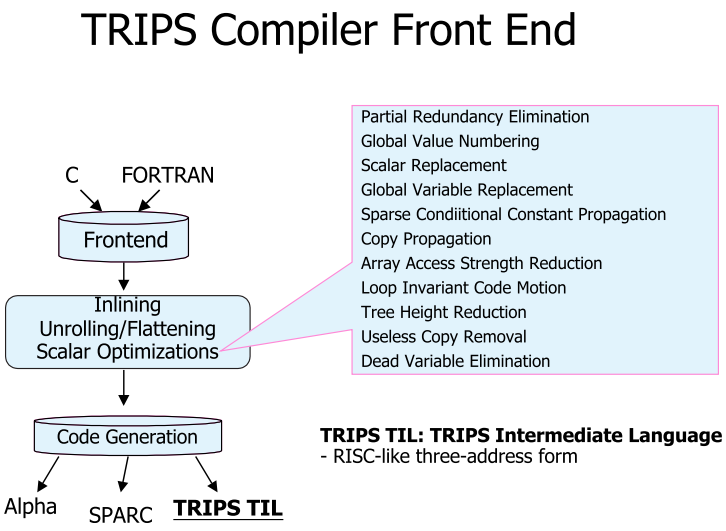


Figure 13: TRIPS Compiler Front End

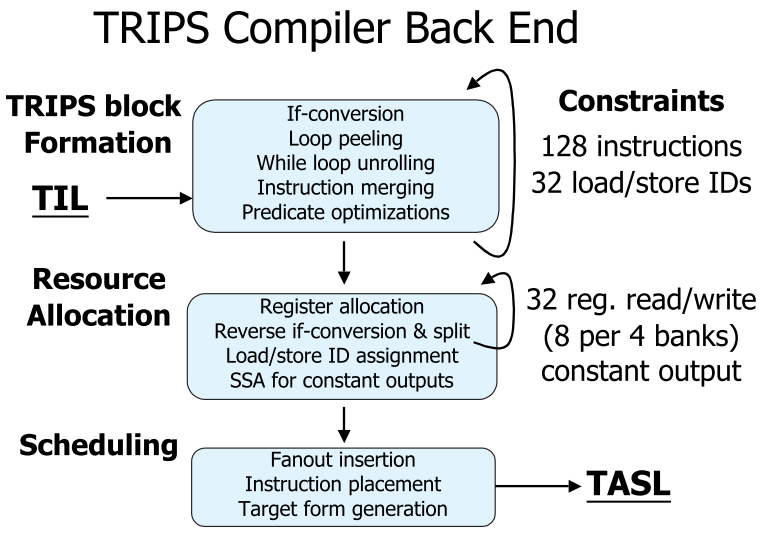


Figure 14: TRIPS Compiler Back End

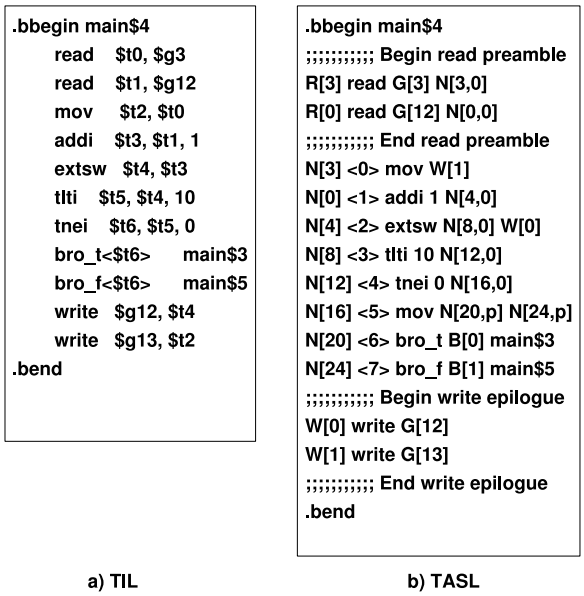


Figure 15: TIL and TASL Example

form has a number of advantages. By using TIL, the compiler is able to delay target form encoding until the very end of compilation, simplifying the backend phases. TIL is easily understood by developers familiar with RISC instructions. It is also possible to translate existing RISC assembly programs to the highest level TIL form and then use the compiler backend to generate TRIPS ISA. This path may be used as a means to port legacy binaries to the TRIPS architecture, a process we prototyped in a proof-of-concept PowerPC to TIL translator.

6.4 Compiler Backend Approach

The challenges of meeting the resource constraints force the TRIPS compiler backend to integrate optimization phases rather than applying them independently, in contrast to compiler structures for conventional architectures. Figure 14 shows that the TRIPS backend compiler phase iterates in several places, whereas compilers for conventional architectures do not. These phases break separation-of-concerns, a key tenet of compiler design that helps simplify the compiler [3]. The first compiler for Fortran and every compiler since then divides the compiler into distinct phases that operate on one or more well-specified intermediate forms. This structure helps simplify each phase which is often heuristically solving an NP-complete problem.

For example, a typical compiler consists of the following phases applied in sequence: parsing, semantic analysis, inlining, alias analysis, loop optimizations, scalar optimizations, scheduling, and register allocation. This structure works best when the upstream phases can ignore the down stream phases. For example, upstream optimizations assume infinite resources such as registers. Consider redundant expression elimination which stores expression results in temporary variables instead of recomputing redundant computation. If this phase cannot assume that all the temporaries go in registers, it must include a heuristic that selects the most profitable replacements. Thus, each phase can be simpler if it assumes the down stream optimization will be able to deal with any resource constraints.

This structure works in classical compilers because only registers and functional units are truly constrained. Therefore, the register allocator goes last, or if the scheduler goes last, it constrains instruction movement such that resulting schedule does not extend live ranges and interfere with the register assignment. This structure enables the other phases to ignore resources, and yet the compiler can satisfy them.

Because of the number of constraints on TRIPS blocks, the TRIPS compiler is not always able to apply the separation-of-concerns principle. For example, the block size constraint and the register constraints conflict. Placing block formation before register allocation requires it to assume an infinite number of registers. If block formation fills a block, but the register allocator needs to spill, then the additional loads and stores may cause the block to violate the block size constraint. If the register allocator goes first, the block formation step must allocate registers each time it merges instructions into a block, which would be prohibitively expensive. The TRIPS compiler resolves these dependencies with judicious use of iteration during register allocation and block formation, which combines basic blocks into TRIPS blocks.

6.5 Satisfying Resource Constraints

The TRIPS architecture constrains blocks in several ways to simplify the implementation of mapping and executing blocks on the distributed hardware. These constraints simplify the hardware, but present a challenge for the compiler. TRIPS blocks are constrained as follows:

1. Block size: Each block consists of up to 128 instructions. The TRIPS prototype has 16 ALUs each with 8 issue slots per block for a total of 128 possible instructions in each fixed-size block. The compiler fills unused instruction slots with no-ops. The compiler thus produces blocks that are aligned in the instruction cache banks, which simplifies mapping the blocks to the ALUs at execution time.
2. Load/Store Accesses: Each block may issue a mixture of up to 32 loads and stores. Each block has 32 load/store (LS) identifiers which are encoded in 5 bits by the individual instructions in a block. This restriction further contributes to a fixed size block encoding. Load (stores) instructions may share identifiers, as long as only one of them issues. Blocks may thus statically contain more than 32 loads and stores, but must dynamically issue at most 32 and only one-per LS identifier.
3. Register Accesses: Each block may read up to 32 registers and each block may write up to 32 registers. These 32 accesses are further constrained to eight reads and eight writes to each of the four banks.
4. Constant Output: Each block must issue a constant number of outputs. Specifically, each block must issue the same number of stores, register writes, and exactly one branch that identifies the next block. The architecture uses these restrictions to determine when the block is finished executing.

While these restrictions are not difficult to meet for small blocks, the TRIPS compiler's goal is to fill each block as much as possible, maximizing the number of instructions in the dynamic window, while adhering to the ISA-specified block constraints. To increase TRIPS blocks beyond single basic blocks and meet the block size and load/store constraints, the compiler extends prior work on predicated hyperblock formation. The compiler uses an algorithm that incrementally combines TRIPS blocks, checking block legality before combination (described in more detail in the next section).

To enforce the load/store constraint and to enforce sequential memory semantics, the compiler assigns a 5-bit load/store ID (LSID) to each load or store instruction to enable the hardware to maintain a total order on memory operations. To mitigate the effect of the limit on LSIDs, and to produce a constant number of outputs, the compiler implements an algorithm that reuses LSIDs on control-independent paths, guaranteeing that only one load or store with a given LSID will fire at runtime. While this problem is NP-hard, our greedy matching algorithm is effective at aligning the loads and stores on independent paths so that LSID allocation is not a bottleneck.

To satisfy the register banking constraints, the compiler employs a form of a linear-scan register allocator. The compiler uses SSA to reveal register writes and stores on disjoint paths, and to guarantee that each path through the TRIPS block produces one and only one write to any register, and one and only one store to a given LSID (the load/store constraint). If any of these constraints are violated (for example, an insertion of spill code during register allocation that causes a block to exceed its LSID limit or instruction count), the compiler splits the offending block using reverse-if conversion, and then re-runs register allocation, iterating as necessary until all TRIPS blocks satisfy the architectural constraints.

These algorithms form the core of the TRIPS compiler technology; they are necessary and sufficient to correctly compile the major desktop and embedded benchmark suites: SPEC2000 and EEMBC. We now discuss the algorithms need to attain high performance for TRIPS.

6.6 TRIPS Block Formation

A key challenge for the TRIPS compiler is to form blocks that meet the constraints and are full of useful instructions. Previous compilers for VLIW architectures formed hyperblocks, which have no structural constraints, but are similar to TRIPS blocks because they are single entry code regions. These compilers selected a discrete ordering for loop unrolling, loop peeling, block formation (if-conversion and predication), and scalar optimizations. However, if block formation occurs before loop unrolling, the compiler will not combine iterations to form larger blocks. If it performs unrolling first, the compiler must predict the effects of block formation to pick an appropriate unroll factor. To form effective blocks, the compiler must balance the interactions among these phases. Although August et al. introduce reverse if-conversion to resolve the phase ordering problem between block formation and scalar optimizations [2], previous compiler research has not focused on the interaction of unrolling and peeling with block formation.

EDGE architectures complicate this phase ordering problem by placing additional structural constraints on blocks as enumerated above. The compiler seeks to fill each block as full as possible to amortize the runtime cost of mapping each fixed-size block to the hardware substrate. If an EDGE compiler conservatively forms blocks that meet the constraints and then applies scalar optimizations that reduce code size, it misses opportunities to pack more instructions into the blocks and better utilize the hardware instruction window.

We introduced convergent block formation to address the challenges of phase ordering and converging to structural constraints. This algorithm incrementally merges basic blocks and repeatedly applies scalar optimizations until it cannot add a new block, thus converging on the limit of the structural constraints. To perform peeling and unrolling, convergent block formation generalizes tail duplication. Tail duplication eliminates a side entrance to an acyclic trace by duplicating code below a merge point. Head duplication eliminates the back edge of a loop (a side entrance to a cyclic trace) by duplicating and predicating the target of the back edge. This process integrates peeling and unrolling with block formation.

This algorithm can implement a wide range of heuristics by carefully selecting the order to merge blocks. VLIW block selection heuristics have focused on minimizing and balancing dependence height, because dependences along any predicate path in a block constrain its static schedule height, even if that path does not execute at runtime. Because the TRIPS microarchitecture supports dynamic instruction issue and allows a block to commit once it produces its outputs, the dependence height of a falsely predicated path has little effect on performance. TRIPS heuristics instead perform best by creating full blocks, removing unpredictable branches, and limiting tail duplication.

We evaluate convergent block formation using simulation and execution on the TRIPS EDGE architecture. These experiments show that this approach improves TRIPS microbenchmark cycle counts by 2% to 11% on average when compared to classical phase orderings. We measure a number of EDGE block policies and also measure VLIW heuristics implemented within the algorithm. Measuring VLIW and EDGE heuristics with and without phase ordering integration shows that integration improves the performance of both by resolving the phase ordering issues of block formation, loop peeling, loop unrolling, and scalar optimizations. Our results also establish a correlation between block count reduction and performance improvement. Convergent block formation substantially reduces block counts and improves performance of the SPEC2000 benchmarks.

6.7 Banked Register Allocation

EDGE and other spatially partitioned processors have non-uniform register access times, which place more burden on the register allocator to consider both bank and register assignment. To optimize for the partitioned layout, the register allocator must consider the location of register banks and data caches, and the placement of instructions on ALUs in order

to decide which register bank to use for each register. For example, if an instruction uses two registers, we prefer those registers to be in the same or nearby register banks in order to reduce the delay. The delay in reading or writing a register depends on the length of the path between the register bank and the ALU that reads or writes the register. Minimizing the communication latencies between partitioned components requires changes to traditional register allocation heuristics.

We developed a bank allocation algorithm for spatially partitioned architectures and evaluated it on the TRIPS hardware [50]. The algorithm is a linear scan allocator that combines register allocation with bank assignment. To limit the number of spills and their cost, the algorithm prioritizes the register assignments based on block frequency and on the number of live ranges used in the block, ultimately assigning registers in priority order. It uses an evaluation function to select a preferred bank, such that register values arriving at the same instruction at the same time are assigned to adjacent or nearby banks. The evaluation function calculates a score for each bank based on previously assigned banks of dependent instructions and the processor's topological characteristics. It then selects an available register from the preferred bank. The algorithm assigns the live range to a register in this bank if the assignment also satisfies the register bank constraints, i.e., each block can only access eight registers from each bank. Otherwise, it goes to a register in the next preferred bank and so on. If no register is available, it spills. We evaluated this algorithm and some variants on the TRIPS hardware. The results show that significant swings in performance are possible, and that bank aware allocation improves over bank oblivious allocation by an average of 6% on the TRIPS hardware.

6.8 Instruction Scheduling

Given the decomposition of a program into blocks of instructions, the EDGE instruction scheduling problem is to assign instructions numbers that determine where on an EDGE processor the instructions in a block will execute. At runtime, the microarchitecture places each instruction on an ALU according to the scheduler's statically assigned number (ID). Each execution tile in the hardware dynamically issues an instruction assigned to it when all of the instruction's operands have arrived. If multiple instructions are ready, the hardware issues lower numbered instructions first. Thus, an EDGE scheduler statically places (SP) each instruction on the ALUs, and the hardware dynamically issues (DI) instructions when their operands are ready. SPDI differs from the VLIW approach, which uses static placement and static issue, and the out-of-order superscalar approach, which uses dynamic placement and dynamic issue. The SPDI execution model creates challenges for the scheduler since it must statically estimate dynamic resource conflicts and critical paths.

A key contribution of the EDGE ISA is that the architecture can interpret these numbers in a variety of ways, making executables portable and independent of the exact ALU topology. However, for ease of exposition, consider the TRIPS processor which contains a four-by-four array of arithmetic units, each one holding up to eight instructions from a 128-instruction block. The TRIPS compiler seeks an assignment of IDs to instructions on this substrate that minimizes the critical path through the block and program. The scheduler balances communication, by placing dependent instructions in proximity, and concurrency, by placing independent instructions on different ALUs. Early in the project, we showed experimentally that the performance of the TRIPS system is sensitive to the scheduling algorithm, with performance varying on microbenchmarks by up to 100% between adversarial bad and good schedules.

6.8.1 Spatial Path Scheduling

We developed two scheduling approaches. We first developed GRST, a GReedy list Scheduling for TRIPS [42]. Dissatisfied with its performance and based on preliminary experimental results with schedules produced by simulated annealing, we continued to improve the scheduler. We subsequently developed a spatial path scheduling algorithm (SPS) [8, 9]. The SPS algorithm exploits the insight that dependent instructions form paths through the block and each path must begin and end with an anchor point, which is a known physical locations for register bank reads/writes, data cache loads/stores, and the global control tile. Figure 12 shows an example of initial anchor points in the TRIPS prototype microarchitecture, which has four register banks above the top row of the four-by-four ALU array, and a column of four data cache banks to the left of the array. There is a chain of dependent instructions that starts with a read to a register in bank 2 (i.e., read r2), computes intermediate values, and then loads from data cache bank 0 (i.e., load D[0]). This chain must traverse at least 4 operand network links, as shown by the path of arrows from R2 to D[0] on the right of Figure 12.

These two locations are anchor points, as are all register, data cache, and global control tile accesses.

SPS exploits anchor points to plan a schedule based on the number of known instructions on a path and their physical locations. SPS proceeds by first fixing known anchor points, and then for all instructions with at least one parent scheduled, it evaluates placing each one at all the available issue slots. SPS computes a cost which estimates when the instruction will execute; lower costs indicate sooner execution. For each instruction, SPS picks the location with the best (least) cost. SPS then examines all the instructions, and chooses the instruction with the highest cost, because that instruction is most critical among the choices to schedule. As instruction are assigned to locations, they also become anchors.

The basic SPS algorithm computes costs based on routing distances using all known anchor points on a path. We augment this basic SPS algorithm with heuristics to model contention on the ALUs and network links, estimate inter-block (global) critical paths, and provide lookahead for planning path routes based on the number and location of instructions on the path. We compared SPS with GRST, the greedy list scheduling algorithm, using a cycle-accurate, validated simulator with hand-optimized kernels drawn from SPEC2000, EEMBC, Livermore Loops, MediaBench, and C libraries. The basic SPS algorithm improved performance by 14% on average and up to 46% over GRST [8]. Similar results hold on the hardware for kernels and larger benchmarks, such as SPEC.

By exploiting anchor points, SPS easily generalizes to many different topologies. One simply provides the microarchitecture and topology in an abstract form; i.e., location, number, and spatial relationship of microarchitectural resources such as PEs, caches, and register files. While we demonstrated SPS's effectiveness for the TRIPS ISA and microarchitecture, we believe it is applicable to schedulers for other partitioned architectures such as WaveScalar [38], and may be useful for clustered VLIWs [20] and RAW [67, 34].

6.8.2 Refining the SPS Cost Function using Learning

A thorough examination of the "ideal" annealed schedules motivated additional modifications to our SPS heuristic cost function [8]. The final SPS algorithm improved performance by 7% over the base SPS algorithm and 21% over GRST. This scheduler is on average within 5% of the annealed schedules. However, we still suspected that additional gains could be obtained.

The central challenge in determining the upper bound for how well the scheduler could perform is that all instruction placements are legal, i.e., given a block of 128 instructions, there are 128 factorial possible instruction placements. Furthermore, the schedule of one block influences the schedule of dependent blocks. These features indicated that simulated annealing might be getting stuck in local minimums, even though it is the best algorithm known for searching large complex spaces.

The complexity of the search space motivated us to investigate if machine learning (ML) could produce better schedules [9]. ML is well suited to learning cost functions, and the core of the SPS function is the heuristic-based cost function that determines the relative cost of an instruction at different locations.

First, we used a systematic feature selection method to explore which characteristics should be used at all in the cost function. Feature selection reduces a feature set size based on the extent to which features affect performance. This exploration resulted in a few new features and the elimination of some redundant features compared to our original SPS cost function.

Then, we then use these features as input to a reinforcement learning technique, called Neuro-Evolution of Augmenting Topologies (NEAT) to generate cost functions. NEAT uses a genetic algorithm to evolve neural networks [64]. We trained NEAT on each benchmark on the TRIPS hardware to determine best schedules for each individual benchmark, executing thousands of experiments per benchmark. When we compare to simulated annealing applied to SPS schedules for individual benchmarks, we found that NEAT outperforms these schedules by on average of 4% and up to around 15%. When simulated annealing starts with the improved NEAT schedule, performance improves by 2% more on average. Simulated annealing does not produce a cost function, it only perturbs a schedule for a single benchmark, whereas the NEAT scheduler is producing cost functions that can schedule more than one benchmark. Unfortunately, NEAT yields a

better schedule for only individual benchmarks and not a general purpose cost function when compared to our highly tuned SPS scheduler.

Finally, we explored if we could classify the benchmarks into categories and produce category specific schedulers with NEAT. We tried an hierarchical approach to machine learning that classifies segments of code with similar characteristics and then learns heuristics for these classes. This approach performs closer to the specialized heuristics, but still leaves performance on the floor. We believe that the classification approach is very promising and should continue to be pursued.

6.9 Summary

To solve the problem of compiling for EDGE architectures, we developed innovative robust compiler technologies and implemented them with a small team, that on average consisted of one staff programmer and three to four graduate students. Our results show that whereas pure dataflow machines required specialized languages, the hybrid dataflow, block-atomic execution model in EDGE architectures enables us to build a compiler for standard imperative programming languages such as C and Fortran. Section 9 presents a detailed performance evaluation that demonstrates the compiler meets its goals of correctness and performance across a wide variety of benchmarks. These results also indicate that there is room to further improve the compiler to match the performance of hand-coded programs. Our experiences and analysis of the performance gaps indicate that attainable improvements in block formation policies, scalar optimizations/instruction code quality, and predication can close this gap.

7 TRIPS Software Development Kit

The TRIPS software development kit consists of a compilation toolchain, a runtime system, a system monitor, debugging tools, and performance tools. The toolchain is responsible for consuming source files written in ANSI C and Fortran 77; applying classic and EDGE optimizations, code generation, and scheduling; and generating binary objects that can be linked with standard math and C runtime libraries. The runtime system software is responsible for interfacing between the TRIPS chip, the board, and the PowerPC host processor. The entire software suite was designed with a set of clean interfaces and as much as possible we leveraged off of existing open source software and the GNU toolchain. While Section 6 described the compiler in detail, this section describes the rest of the software development environment for the TRIPS architecture.

7.1 Runtime Libraries

To provide a complete working system, we developed a set of standard C runtime libraries, a math library, and a set of optimized libraries tuned to the ISA's unique features.

C runtime library: We chose the Dietlibc embedded C runtime library for its small footprint, base set of functionality, and portability [69]. We implemented two main TRIPS-specific customizations. First, we implemented a generic system call interface, so that the compiler treats system calls as function calls whose definitions are automatically generated by the `m4` macro preprocessor with register setups, traps to the `SCALL` instruction, and return values. Second, we defined routines in the C runtime startup module (`crt.o`) to interface with the program loader and set up the stack, provide software floating point division, determine the base physical address of the chip configuration space, implement `setjmp()`, and so on. Finally, we increased the amount of buffering used by `printf()` and the memory manager in calls to `malloc()`, to minimize runtime overhead.

Math libraries: We chose the SunSoft Freely Distributable LIBM (libfdm) C math library for its transcendental math functions [36]. The libfdm routines are IEEE-754 conformant, portable, and well-tested. Since the prototype lacks a floating-point divide unit and no `FDIV` instruction, we implemented a software divide routines derived from a platform-independent implementation of the IEEE Standard for Binary Floating-Point Arithmetic [25]. We then replaced key functions such as `sqrt()` with those from the IBM IEEE math library (MathLib) [19], as this library uses an accurate table method and executes efficiently on the prototype.

Optimized string and memory library: The dietlibc runtime library is designed to be compact and portable across platforms but not necessarily optimal for a given platform. For example, the dietlibc `strcpy()` routine has at its kernel:

```
while (*dest++=*t++);
```

This implementation is portable and reasonably efficient, but can be re-written to take advantage of large blocks of predicated instructions. We developed further optimized compiler-generated TIL-coded for `strcpy()` and other key string and memory routines and grouped them into an optimized library that the linker consults before falling back to the corresponding `dietlibc` routine.

7.2 Binary Utilities

We ported many of the GNU binary utilities (binutils), available from the Free Software Foundation [21] to TRIPS. Examples of easily ported programs include the assembler, linker, and other utilities, such as `nm`, `objdump`, and `ar`. We

decided to port the GNU binutils for three main reasons: 1) they provide a rich and mature codebase to start with, 2) a large and active experienced development community exists, and 3) a common set of application programming interfaces are provided which can be used by other tools, such as the simulators and debugger. We chose to support a simple version of the Executable Linking Format (ELF) to output a small number of string tables and text, data, and bss program sections. All of the TRIPS calling conventions are detailed in the Application Binary Interface (ABI) manual [62].

All TRIPS executables are statically linked (no shared libraries), which dramatically simplifies installing, versioning, and debugging toolchain components and applications. The utility frontends are largely platform independent. For example, once we specified the TRIPS data types, such as 64-bit pointers, longs, and doubles and 32-bit integers and floats, the GNU utilities transparently provided support for allocating and accessing TRIPS scalar variables. However, to support the prototype ISA and high-level compiler the TRIPS-specific back ends required a significant porting effort as described below:

Block-oriented ISA: The GNU frontend assumes that each line of a source file translates into one instruction word. Once the TRIPS Assembly Language (TASL) had been defined, encoding the individual 32-bit instructions proved straightforward, simply requiring the `tas` assembler to parse and translate the source file a line at a time [72].

However, to support the TRIPS block-atomic model, TASL syntax groups instructions and register reads and writes into blocks, demarcated with "block begin" (`.bbegin`) and "block end" (`.bend`) directives for that block. The `tas` assembler therefore tracks the individual statements after the opening `.bbegin` directive and enters them into a memory structure representing the execution grid and architectural registers. When encountering the matching `.bend` directive, the assembler traverses the data structure, marks instructions that require relocation, uses the GNU Binary File Description (BFD) routines to translate x86 Little-Endian formats to TRIPS Big-Endian, and commits the whole block to disk.

Throughout binutils development, a balance needed to be struck between treating instructions as individual 32-bit words and treating "instructions" as variable-size code blocks. In addition, the ISA definition of 32-bit instructions and 64-bit pointers and data types required syntactical support to construct 64-bit data from pieces of the 32-bit instructions.

TRIPS binary disassembly: The binary utilities and debugger provide several disassembly routines. Again, the GNU model of one instruction per code word had to be extended to support several TRIPS or EDGE specific features: consuming a header chunk of 128 bytes at once; parsing and rearranging the header into register reads and writes; reading the following one to four instruction chunks; and disassembling the individual code words, while maintaining integrity of the complete code block.

7.3 Software Simulators and Debuggers

We have developed a variety of software development tools for TRIPS, including a simple ISA emulator, a detailed cycle-accurate simulator, a critical path analysis tool, and a debugger. Despite disparate goals these tools share the same C++ code base and key attributes:

- A built-in program loader, using the *BFD* utility routines
- Support for `argv`, `argc`, and `envp` program variables and for `stdio`, `stdout`, and `stderr` console I/O
- A common execution model of fetching, map, and execute
- A set of proxy services for handling system calls
- Common tracing and debugging output formats
- A set of statistics collection routines.

7.3.1 Functional Simulator

The `tsim_arch` functional simulator is an architecture-level simulator intended to accurately model the TRIPS processor architecture. It is equivalent to a traditional instruction set simulator (or functional emulator) and does not provide realistic cycle counts.

7.3.2 Timing Simulator

In addition to simulating TRIPS processor functionality, `tsim_proc` offers a detailed, cycle-accurate model of the prototype TRIPS processor. It models the internal organization and latencies of the processor and is intended to support processor-level performance analysis.

Whereas `tsim_arch` models the behavior of architectural registers and execution nodes at the software-visible level, `tsim_proc` additionally models the microarchitecture structures within each. For example, `tsim_proc` details the behavior of individual execution nodes when sent commands from the global control unit, when reading packets from and writing packets to the operand network, when filling and consuming internal buffers, and during block flushes and commits–in addition to actually selecting, executing, and retiring instructions. During performance tuning after RTL development, `tsim_proc` was validated and tuned to be cycle accurate within 2% of the actual TRIPS hardware. On a 2.9GHz Pentium 4, `tsim_arch` can execute 800,000+ simulated instructions per second. The increased level of detail causes `tsim_proc` to simulate 300-800 times slower than `tsim_arch`.

7.3.3 System Simulator

A system simulator (`tsim_sys`) was also developed to simulate execution of multiple processors, enabling the user to download and execute one or more applications under the control of the TRIPS Resource Manager, discussed in section 7.4.

7.3.4 System call support

All simulators provide limited system call support using a common interface and codebase, driven by the requirements of targeted applications. Currently, the simulators support the following: `brk()`, `close()`, `creat()`, `exit()`, `fstat()`, `getpid()`, `gettimeofday()`, `lseek()`, `lstat()`, `open()`, `read()`, `stat()`, `time()`, `unlink()`, and `write()`. The C runtime library translates system calls into `SCALL` instructions, which trap into the simulator's system services module. The trap handler in turn proxies the service request on the host, according to the POSIX definition of the call.

7.3.5 Critical path analysis

Similar to a modern processor, TRIPS exploits fine-grained concurrency from potentially hundreds of instructions in flight. Each instruction passes through a myriad of hardware resources resulting in numerous microarchitectural events during the course of its lifetime—cache misses, branch mispredictions, re-order buffer stalls, port contentions, and data dependence hazards. We developed a critical path analysis tool called `tsim_critical` based on the methodology proposed by Fields et al. [18, 68].

Compared to previous work that focused on superscalar processors, critical path analysis for TRIPS is more complex because of the larger number of microarchitecture events, distributed nature of processor control, and the significantly larger number of instructions in flight. Details of the tool are described in [41]. We used this tool extensively in our performance analysis and in validating `tsim_proc` against real hardware.

7.3.6 Debugger

We developed a TRIPS symbolic debugger, `tdb` similar to gdb that supports TRIPS-specific data types, breakpointing, block stepping, memory and register accesses, and runtime libraries.

Due to its powerful feature set and familiar interface, we chose to port the GNU debugger `gdb` to the prototype. We used `tdb` to debug applications running on the TRIPS chip while `tdb` itself runs on host. The GNU Dynamic Data Debugger (`ddd`) has been ported to the prototype system as a frontend to `tdb` [10]. By providing a friendly user interface and scrollable visual listings, `ddd` enables users to orient themselves in their source code while stepping through their application. Those portions of `tdb` requiring further customizations are discussed below.

Host-target protocol: `gdb` normally offers support for remote debugging in two forms. First, a platform-specific stub library is linked into each target executable and handles interrupts and commands from the host. Second, a `gdbserver` runs as a proxy on the target as a separate, heavy-weight process and controls the behavior of the target application.

Both forms use a low-level string-based protocol between host and target. We defined a higher-level protocol and an API to communicate between (a) the `gdb` backend on the Host PC and (b) the EBI on the motherboard. The API includes routines to download and execute programs, to set and clear breakpoints, to continue and block-step execution, to read registers and virtual memory locations, and to map virtual to physical addresses.

Symbol table entries: The binary utilities and `gdb` frontend already provides extensive support for symbol table, data type, and line number information in the form of stabs entries. However, the TRIPS compiler required extensive effort to generate symbol and line information.

Breakpoints: The processor offers both break-before and break-after hardware breakpoints on a per-block basis. The TRIPS system includes a TRIPS Resource Manager (TRM) that acts as a rudimentary runtime system to provide services to users and applications. The debugger employs TRM functions to set and clear both types as well as GNU platform-independent functions. `tdb` uses a mix of functions for its breakpointing support. When execution has stopped on a breakpoint and the user issues a continue command, `tdb` uses TRM functions to remove the current break-before breakpoint, set a break-after breakpoint, and continue execution. When it almost immediately hits the break-after breakpoint, `tdb` uses TRM functions to restore the original break-before breakpoint, clear the current break-after breakpoint, and resume execution transparently. These features allow the user to interact with the debugger in familiar fashion.

7.4 System Software

This section describes the TRIPS system software as it executes on the prototype hardware. However, much of this same software executes equivalently on our system simulator platform. System software goals include enabling efficient hardware bring-up; enabling users to download, execute, observe, manage, and debug applications; demonstrating the performance and capabilities of the prototype on targeted workloads, such as on EEMBC and SPEC CPU2000 benchmarks; and supporting full hardware utilization of processor, chip, memory, and board resources.

7.4.1 System software components

Figure 16 shows the components of the TRIPS system software which include:

- An x86/Linux Host PC to manage the motherboards and provide a networked file system

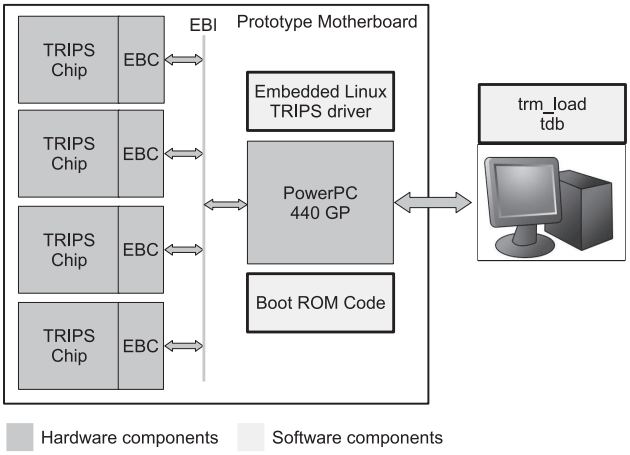


Figure 16: System Software Components

- A local area network to support communication among the motherboards and Host PC
- A hardware debugger to provide initial access to the PowerPC 440GP registers, TLB entries, peripheral bus, SDRAM controller; and to program the onboard flash memory
- A bootloader programmed into flash memory
- An embedded Linux kernel executing on the motherboard
- A device driver and daemon process to mediate communication between motherboard and target processors
- A cross-compilation toolchain for the embedded PowerPC 440GP processor
- A resource manager running on the Host PC to monitor and control the target processors.

To manage complexity, system software is organized in layers, from device driver on the motherboard to end-user clients on the Host PC. To leverage engineering resources, much of the software derives from Open Source and off-the-shelf products.

7.4.2 Board Software

Up to four dual-core TRIPS chips on individual daughtercards populate the prototype motherboard. The embedded PowerPC 440GP processor serves two major functions. First, it reads and writes addresses in the global memory space shared among processors. Second, it communicates with the External Bus Control unit on each of the four TRIPS chips to handle TRIPS system calls.

Boot process: By adopting the Open Source `u-boot` bootloader and Linux 2.6 kernel tree, we greatly mitigated the complexities of booting the PowerPC 440GP into Linux [11]. During a motherboard reboot, the PowerPC 440GP is reset and begins executing `u-boot` from flash memory to initialize processor registers and peripherals, including its UART, Ethernet, and SDRAM controllers. After copying itself to SDRAM and jumping there, the bootloader communicates across the local area network (LAN) to the Host PC to download the kernel into processor memory and transfer execution to the kernel entry point. After re-initializing board components, the kernel communicates across the LAN to learn its hostname and network parameters (via DHCP), to set the onboard clock (via NTP), to mount its root file system (via NFS), and to accept logins (via SSHD).

EBI adapter: At the end of boot process, a boot script creates a `/dev/trips` device file, installs the `ebi_driver.ko` kernel module, and invokes the `ebi_adapter` user-level process.

After detecting which TRIPS chips are physically present on the board, the EBI adapter spawns two threads–one to listen on a pre-defined socket for Host PC commands from the TRM and another thread to wait on processor interrupts potentially from all four chips.

7.4.3 Resource Manager

The TRIPS Resource Manager, a user-level process on the Host PC, functions as a light-weight operating system, initializing the chips, allocating memory, downloading applications, servicing system calls, and monitoring performance. The prototype relies on the Host PC for all system services, including console and file i/o, memory allocation, wallclock access, and process termination.

We defined a low-level Hardware Abstraction Level (HAL) consisting of a protocol and API supported between the TRM executing on the Host PC and the EBI adapter executing on each motherboard. The protocol is simple but powerful, consisting of individual read and write requests, which always originate from the TRM, and a signalling mechanism by which the EBI adapter can notify the TRM of processor exceptions.

7.5 Summary

The TRIPS software development kit is quite mature for an experimental machine. The elements of the toolchain are built using familiar interfaces and are generally easy to use by programmers. The SDK has been used not only in house at UT-Austin, but also at the Air Force Research Laboratory and at ISI-East. The SDK has also been made available to other researchers; the tools have been downloaded over 65 times from our web site.

The SDK development process was quite streamlined, with little redesign and rework of developed code. Developing our suite of simulators with a common shared code base dramatically reduced code development replication. `tsim_arch` directly served as our reference simulator for hardware verification, and `tsim_proc` was used as our initial hardware performance validation tool. The common code based and single interface also proved very useful during the system bringup process. In fact, we were able to develop and bring up nearly all of the system software in advance of the hardware being completed. We leveraged as much as possible from existing open source software, which was highly effective for the binary utilities, debugger, open source runtime libraries, and system software. We were quite impressed by the overall quality and support for these open source libraries.

8 Software Support for Parallelism

TRIPS is a parallel system that includes two processors per chip and is scalable up to 32 chips (64-processors). While much of our work has focused on fine-grained data-driven parallelism within a processor, we have also built tools and applications for executing large scale multithreaded parallel programs on TRIPS. This section describes two of our software efforts that have built support for parallel programming. Section 8.1 describes our design and implementation of a Message Passing Interface (MPI) library that exploits configurable features of the TRIPS hardware. Section 8.2 describes the R-Stream compiler built by Reservoir Labs that was funded in part through the TRIPS contract. R-Stream is designed to automatically extract stream-oriented parallelism from high-level programs without requiring the programmer to orchestrate any threads.

8.1 Message Passing Interface

MPI is a standardized message passing library specification. While many portable MPI implementations exist, it is important to make architecture specific tuning to the MPI library for good performance. The TRIPS block based execution model creates new challenges for message passing and requires new solutions and optimizations to achieve good performance. For this reason, we decided to implement an MPI library from scratch for TRIPS. This section provides a short overview of an MPI that employs the DMA controllers and the configurable memory of TRIPS. Additional details can be found in [33]. The MPI library is also used to construct the parallel applications described in Section 9.

Figure 17 shows the organization of an MPI library on TRIPS. MPI applications link to the MPI library for communication routines. The MPI library implements collective communication routines using point-to-point routines. The point-to-point routines themselves are implemented using blocking and non-blocking communication routines. Finally, all communication routines use internal implementations of communicators and message queues to implement message passing. The MPI library uses the TRIPS DMA device driver and synchronization library (mutex) for data transfer and locking mechanisms. At the lowest level of the stack is the TRIPS system software and the TRIPS hardware.

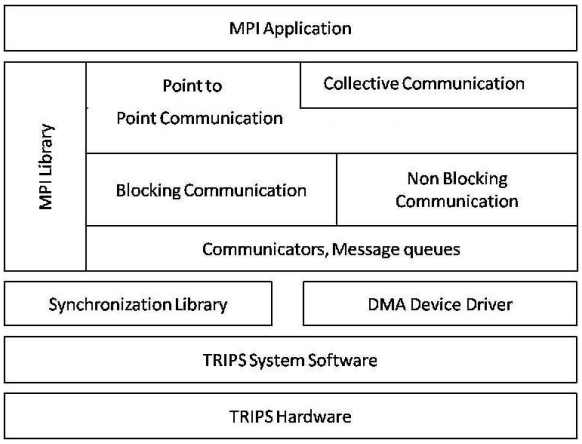


Figure 17: MPI on the TRIPS System Software Stack

8.1.1 Simplifications to MPI library

The full MPI specification is quite large but only a small subset of calls are used in most MPI programs. Since we sought to analyze the efficiency of message passing on TRIPS, we only implemented the subset of the MPI functions required to execute our set of benchmarks and applications. Most applications that use the full library can be modified to use our

Table 6: Comparison of L2 and SRF Modes

Characteristics	SRF mode	L2 mode
Scalability	High - any number of processes	Limited by TLB entries (Up to 13 processes)
Space	256K per process	Limited by SDRAM
Cache behavior	1) No misses for message passing data structures 2) More capacity and conflicts misses for application data (smaller L2)	1) Possible capacity and conflict misses for message passing data structures 2) Flexible cache usage between message passing and application data

library without losing any application functionality. The simplifications we make are universal and not tied to the MPI library on TRIPS. We made the following simplifications to the MPI implementation:

- No support for new communicator creation, group creation and process topology. Processes have access to only the default communicator - `MPI_COMM_WORLD`.
- No support for buffered, ready and synchronous mode of sends and receives.
- No support for creating new MPI data types
- No support for profiling

We implement only the following functions: (1) `MPI_Init` and `MPI_Finalize`, (2) blocking and non-blocking send and receive functions, (3) collective communication, (4) barrier operation and (5) helper functions to get various environment values and return status of functions.

8.1.2 Message passing on TRIPS

TRIPS has a global address space system with physically distributed memory. On a global address space system, inter-process communication can be achieved using shared memory segments. In this section, we outline the design of MPI on such a global address space system. The communicating processes share a memory location to exchange information. The shared memory segment holds internal message passing data structures like message queues to enable exchange of control information. It also holds space to buffer messages temporarily between send and receive routines. Data can be transfered between MPI process using the TRIPS processor or the TRIPS DMA. Using the TRIPS processor for transferring data between MPI processes is a two step process. In the first step, the sender process copies the data from its buffer to the shared segment buffer. In the second step, the receiver process copies the data from the shared segment buffer to its buffer. Using the TRIPS DMA, data can be transferred with a zero copy mechanism. The TRIPS DMA can move data directly from one process's address space to another, bypassing the virtual memory system.

8.1.3 Placement of Message Queues

Physical location of shared memory-segments: TRIPS enables two modes of mapping shared segments: (1) L2 mode and, (2) SRF (streaming register file or scratchpad) mode. In L2, mode, the shared segment is mapped to cached SDRAM region. Since, TRIPS caches are not coherent, the shared segment must be marked as L1 uncacheable to ensure correctness. In this mode, the shared segment data will be in M-tiles configured as L2 cache or in SDRAM. In SRF mode, the shared segment data will be in M-tiles configured as SRF. Table 6 summarizes the tradeoffs between the L2 mode and the SRF mode.

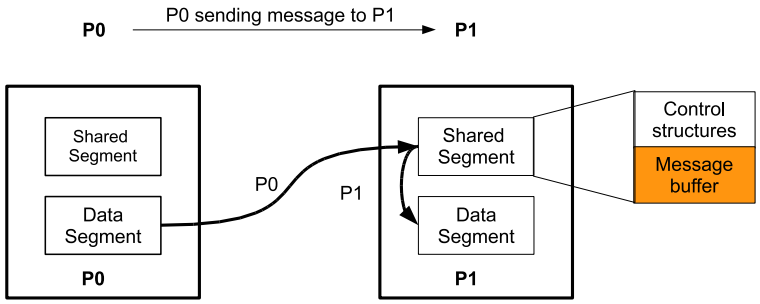


Figure 18: Transferring Data with Buffering in a Shared Segment (Using `memcopy`)

Number of shared segments: TRIPS has physically distributed memory. The following two designs are explored based on the number of shared segments created and where they are logically placed.

1. Centralized shared segment: All MPI processes use only one shared segment. The shared segment can be physically located in any processor's memory. Since process 0 does most of the initialization, the shared segment is mapped to the memory of the same processor as process 0. The advantage of this design is that it is simple to implement and it requires just one TLB entry for each process. The disadvantage is that it has a huge performance penalty due to network congestion since every process has to communicate through this one single memory segment. For example, if process 1 must communicate with process 2, it must go through process 0's memory.
2. Distributed shared segment: Each MPI process has a separate shared segment physically located in its own processor's memory. Each process requires one TLB entry per shared memory segment. This design has a performance advantage over the first design since it reduces network traffic.

Even though the first round of implementation of MPI on TRIPS used centralized shared segment, we replaced it with distributed shared segments because of the scalability and performance of distributed shared segments.

8.1.4 Data Transfer Modes

Buffered mode: In this mode, the sender copies the data from its data segment to the message buffer and encodes the address of the message buffer entry in the header. When a matching receive is posted, the receiver copies the data from the message buffer to its data segment. Figure 18 shows this process. Two copy operations are involved in transferring data. Since the processor is involved in copying the message, there can be no overlap of communication and computation. The advantage of this mode is that it requires no synchronization or handshake between send and receive calls. The send can complete even before the matching receive is posted.

Zero-copy mode: In this mode, a message is copied directly from the source's data segment to the destination's data segment via physical address exchange using DMA. Figure 19 shows this process. If send is posted first, it posts the physical address of the data to be sent in the message queue header. When a matching receive is posted, it translates the virtual address of the message to be received to physical address and programs the DMA to copy the data from source physical address to destination physical address. If receive is posted first, it posts the physical address of the destination message to the prepost queue. When a matching send is posted, it translates the virtual address of the message to be sent to the physical address and programs the DMA to copy the data from the source physical address to the destination physical address. This mode uses a zero copy mechanism but requires explicit synchronization between send and receive

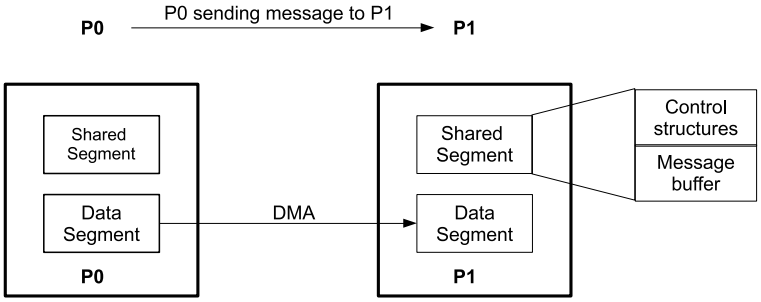


Figure 19: Transferring Data without Buffering in a Shared Segment (Using DMA)

Table 7: MPI Latency Comparison

Communication Type	**L2 Mode** (Cycles)	**SRF Mode** (Cycles)	**InfiniPath** (Cycles)
Intra-chip (P0 and P1)	1412	1453	3432
Inter-chip (C0 and C1)	1688	1813	NA

process for physical address exchange. For this reason, the send cannot complete before the matching receive is posted, thereby forcing it to operate in synchronous mode. Since the DMA controller performs the data transfer, the processor can proceed to other computation, thereby overlapping communication and computation. This mode enables the implementation of non-blocking communication.

8.1.5 Design Evaluation

Latency: We measured the base latency and bandwidth achievable with different versions of the TRIPS MPI library to examine the contributions of using the TRIPS configurable hardware. Table 7 shows the average MPI latency for a round-trip message between two processors on the same chip and on different chips. Our analyses of hardware counters show that the higher MPI latency in SRF mode is due to (a) higher access latency of data in SRF, and (b) higher conflict miss rate in the L2 caches in the SRF mode as compared to the L2 mode. Although the MPI latency benchmark executes with only 50% of the L2 cache in SRF mode as compared to L2 mode, once the caches are warmed up, the L2 mode and the SRF mode will have similar cache miss rates since the program has a very small memory footprint. However, in SRF mode, we configure the M-tiles groups farther from the processor as SRF to hold the shared segment and M-tile groups closer to processor as L2 cache to hold the application data. In L2 mode, all the banks are configured as L2 cache and banks closer to the processor can hold both the shared segment and application data.

We compare our results against an MPI implementation on InfiniBand by Pathscale called InfiniPath [14]. This reference implementation is the best available implementation to the best of our knowledge. As seen from the table, the optimized TRIPS MPI library performs better than the reference implementation. TRIPS MPI achieves such low latency because: (1) on-chip MPI control structures, (2) user accessible light-weight network for communication, and (3) absence of system calls in send and receive routines.

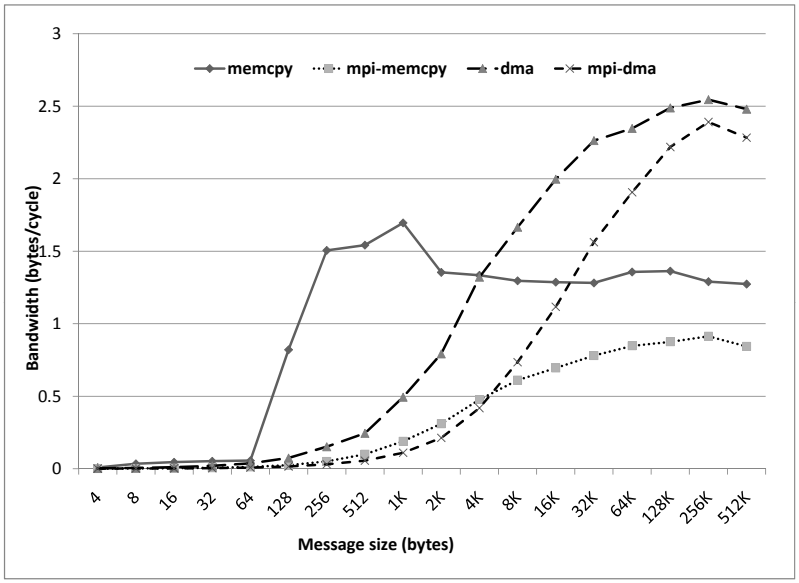


Figure 20: MPI Bandwidth

Bandwidth: To examine the bandwidth potential of the TRIPS mechanisms, we implemented a microbenchmark that performs sends/receives in which the message size can be varied. The microbenchmark is executed with two versions of the MPI library: (1) `MPI_memcopy` which uses `memcopy` (optimized memory copy performed by the TRIPS processor) routine to transfer data between send and receive calls using message buffer, (2) `MPI_DMA` which uses DMA to transfer data without any buffering during send and receive calls. We compare the results to the raw `memcopy` and DMA bandwidth of the system.

Figure 20 shows the achieved bandwidth as a function of message size, which shows the effect of message overhead. The four curves correspond the bandwidth achieved using `memcopy`, `MPI_memcopy`, DMA, and `MPI_DMA`. Compared to a peak bandwidth of 3.2 bytes per cycle across the chip-to-chip (C2C) interface, the bandwidth using `memcopy` achieves a peak of 1.6 bytes per cycle at a message size of 1KB but saturates at 1.4 bytes per cycle for larger messages. The maximum bandwidth achieved by `MPI_memcopy` is 0.9 bytes per cycle at a message size of 128KB. The raw DMA bandwidth saturates at 2.5 bytes per cycle which is lower than the the theoretical C2C peak by 22%. The bandwidth of MPI using DMA saturates at 2.4 bytes per cycle. Thus, MPI bandwidth using DMA follows the raw DMA bandwidth very closely, demonstrating that the MPI implementation is lightweight and efficient.

To achieve the best performance, our MPI implementation selects how to transfer the data, based on message size. The MPI library is tuned for three message sizes: (1) small messages (8 bytes or less) which correspond to the size of a large number of MPI messages [16], (2) medium sized messages (between 8 bytes and 16 KBytes), and (3) large messages (greater than 16 KBytes). Small messages can be encoded within the MPI message header. Medium sized messages use `memcopy` to keep the message initiation overhead down. Large messages use DMA to benefit from the highest bandwidth.

8.1.6 Summary

Our results show that MPI on TRIPS system achieves better latency than a state of the art implementation but not as small as expected from a system with on-chip resources and user accessible hardware. The latency is larger than expected because locks are quite expensive in TRIPS. Our MPI optimizations that reduce the number of locks used during communication phase achieves much lower latency. MPI bandwidth closely follows raw bandwidth of the system proving that the MPI implementation is lightweight and does not add significant overhead to the message transfer. Thus, the

TRIPS primitives were good for moving large amounts of data but the overheads are still high for small messages. Using the different modes of the TRIPS configurable memory (SRF versus L2) results in performance advantages and disadvantages on real applications. In addition to the latency and bandwidth experiments, we also ran five of the NAS parallel benchmarks on the TRIPS hardware. We found that one application benefited from SRF mode because its frequent communication caused interference between the message queues and program data in L2 mode. However, a different application suffered worse performance in SRF mode because the large data footprint during the computation phase caused more L2 cache misses when the on-chip memory is carved into separate but smaller L2 caches and message buffers. We believe that configurable memory will be valuable for future systems, but that some of the design simplifications of TRIPS (such as no coherence between L1 and L2) make it hard to demonstrate on the prototype.

8.2 Stream Compilation for TRIPS

As part of the TRIPS Implementation effort, Reservoir Labs, Inc. has developed a high-level compiler (HLC) that exposes program concurrency to be exploited across multiple TRIPS cores and processors. This compiler, called R-Stream[35], provides a means for programmers to obtain this concurrency automatically from sequential programs expressed in C. In addition to achieving high-level concurrency, R-Stream can simultaneously restructure programs so that greater degrees of low-level concurrency are available for the TRIPS low-level compiler (LLC) to parallelize across the distributed microarchitecture of the TRIPS core.

The TRIPS architecture achieves high performance and scalability by providing mechanisms that allow for explicit management of hardware by software. For example, TRIPS avoids implementing a complex strict memory consistency model in hardware in favor of allowing software consistency management. TRIPS provides DMA operations and messaging mechanisms to communicate in bulk among distributed memories and processors. Bulk communication allows for greater power efficiency and throughput by amortizing the overheads of communications and by allowing software to read or write only the specific sections of main memory needed by the algorithm. Thus bulk communication is more efficient than the traditional memory access mechanism of hardware managed caches, which typically read and write redundant information through extra words in cache lines or through write-backs of dirty, but unneeded data. The high-level transformations performed by the R-Stream compiler result in mapped code that meets these software responsibilities to achieve the high potential performance of multi-TRIPS systems.

8.2.1 R-Stream Overview

R-Stream accepts algorithms written in sequential ANSI C89 within a form called a “static control program” [17]. R-Stream provides powerful analysis of the algorithm and reshapes it into a schedule in space (where things happen over processors and memories) and time (when things happen) to optimize performance. R-Stream transforms the sequential input description of the algorithm into a streaming execution model, with explicit high-level concurrency, improved locality of reference, and explicit control of architectural features. Programmer productivity is improved; programmers express their algorithm in familiar sequential programming abstractions and use R-Stream’s automatic mapping capability to get good hardware performance.

The streaming execution model reflects the explicit control of the hardware. In this model, small finite tasks are executed that use and produce a small set of data, reading and writing from local memories. These tasks execute concurrently and in a high-level pipelined manner: bulk communication to fetch the input data for the next task executes concurrently with current task, while bulk communication for the previous results are sent back to main memory. This software-pipeline is explicitly expressed in the code produced by R-Stream.

The enabling technology in R-Stream is a program representation and set of mathematical tools for analysis and program transformation based on the the mathematics of polyhedra.[1] The R-Stream compiler represents iteration spaces, array access functions, and dependencies as multi-dimensional parametric polyhedra. When in this form, the compiler problems

[1] More specifically, Z-Polyhedra, the intersection of integer lattices with polyhedra.

of solving for mappings can be concisely stated and solved in a single representation using common mathematical optimization libraries. These solved problems include (1) providing fine-grained and coarse-grained parallelism in pipeline, data, and task form; (2) balancing locality; (3) grouping related iterations into “tasks;” and (4) restructuring the code into software pipelines.

R-Stream is also engineered on a solid base infrastructure. The infrastructure starts with a commercial C front end from Edison Design Group (EDG), and a scalar program infrastructure that uses a graph based static single assignment form that combines some aspects of value dependence graphs (VDG). C syntactic types are combined and represented explicitly according to rules resembling a simply typed lambda calculus. This scalar representation allows the compiler to maintain the semantics of the original input program strictly, yet independently of syntactic artifacts in the original source. Further, and uniquely, they allow R-Stream to emit the resulting mapped program in terms of code syntax idioms that resemble human authorship. This facilitates the critical downstream job of downstream LLCs that must achieve good compilation of the low-level tasks. We have discovered that most LLCs, including the TRIPS LLC, key on specific human code idioms to achieve their optimization.

8.2.2 R-Stream Compilation Strategy for TRIPS

For TRIPS, R-Stream maps computationally-intense parts of the application that adhere to the extended static control program form to multiple TRIPS processors. The R-Stream mapping process consists of a series of separate mapping phases. The phases are as follows: (1) expand arrays to reduce false dependencies, (2) find parallelism balanced with spatio-temporal locality, (3) group iterations into “tiles,” (4) place computations physically on processors, (5) improve locality by allocating local memories, (6) generate explicit communications, (7) refine communications to reshape subsets of arrays into more compact form in local memory, and (8) form high-level software pipelines. R-Stream emits the schedule that results from this process as a new, restructured C source in terms of a simple execution abstraction.

This C output for TRIPS is parallelized. It accesses the features of the TRIPS architecture for messaging, task initiation, and synchronization through a thin runtime. This thin runtime abstracts features of the TRIPS architecture to an execution model similar to other architectures that R-Stream targets.

8.2.3 TRIPS Machine Model

The R-Stream polyhedral mapper requires certain information about a target in order to make and frame decisions on how to reshape code and optimize it for the target machine. The programmer encodes this information in a machine model that is specific to each target for R-Stream.

For example, as mentioned above, the execution model for TRIPS is a streaming model with TRIPS processing elements passing data among themselves using DMA operations. Thus, the TRIPS machine model provides a model of the DMA operations that the R-Stream mapper uses to effect streaming. The TRIPS machine model for DMA encodes the following properties: minimum block size, preferred block size, maximum stride size, maximum number of strides, and overhead vs. bandwidth for strided and indexed DMA. The R-Stream mapper uses the values of these properties to compare choices about how to parallelize across the TRIPS processors in such a way that DMA communications are efficient.

While the R-Stream mapper has the ability to target a heterogeneous target (e.g., IBM Cell), for TRIPS the target is (at present, at least on the current TRIPS board) multiple identical processors. Also, in TRIPS, the processors can issue communication requests and can participate in synchronizations. The machine model encodes these properties too. With this outline, the mapper produces code where one TRIPS processor is the surrogate master processor, which issues commands to initiate computation on itself and other TRIPS processors, and which then handles the synchronization among them.

Other important characteristics for TRIPS that the machine model encodes include local and global memory sizes, cache sizes (total and line), and the geometry of the PEs.

8.2.4 Mapping Matrix Multiplication to TRIPS

To illustrate the R-Stream mapping results for TRIPS, the following code excerpts shows the inputs and the intermediate results of phases of mapping the well-known matrix multiplication kernel to a system with multiple TRIPS processors:[2]

```
double A[128][128];
double B[128][128];
double C[128][128];

...
for (int i = 0; i <= 127; i++) {
  for (int j = 0; j <= 127; j++) {
     C[i][j] = 0;
     for (int k = 0; k <= 127; k++) {
        C[i][j] = C[i][j] + A[i][k] * B[k][j];
     }
  }
}
...
```

After parallelization, each processor will execute the following loop in a Single Program Multiple Data (SPMD) fashion. The `proc_id` processor number offsets the subsets of iteration space so that each processor's assigned subset is non-overlapping.

```
double A[128][128];
double B[128][128];
double C[128][128];

void kernel(int proc_id, ...) {
 // Control here
 for (int i = proc_id; i <= 127; i+=4) {
    for (int j = 0; j <= 127; j++) {
       // Kernel here
       C[i][j] = 0;
       for (int k = 0; k <= 127; k++) {
          C[i][j] = C[i][j] + A[i][k] * B[k][j];
       }
    }
 }
}
```

The main thread executes the kernel by spawning four threads running on four TRIPS processors via the call

```
int main() {
 ...
 TRIPS_mapped_end(TRIPS_mapped_begin(0, 4, 0, kernel, ...));
 ...
}
```

[2] Although the steps are illustrated with C code, the full mapping process actually takes place in the abstract polyhedral form. The code at intermediate stages represents lowered samples of the representation between mapping phases.

While weakly coherent shared memory is available on TRIPS, messaging is more efficient. R-Stream makes the communication explicit through a *communications generation* optimization that inserts DMA operations around the kernel (k-loop). The result looks like this:

```
// Global arrays
double C[128][128];
double A[128][128];
double B[128][128];

void kernel(int proc_id, ...) {
 // Local arrays
 double C_loc;
 double A_loc[128];
 double B_loc[128];

 for (int i = proc_id; i <= 127; i+= 4) {
    for (int j = 0; j <= 127; j++) {
       TRIPS_dma_get(&A[i][0], &A_loc[0], 1024, 8, 8, 1, 0);
       TRIPS_dma_get(&B[0][j], &B_loc[0], 8, 1024, 8, 128, 0);
       TRIPS_dma_wait(0);
       C_loc = 0;
       for (int k = 0; k <= 127; k++) {
          C_loc = C_loc + A_loc[k] * B_loc[k];
       }
       TRIPS_dma_put(&C_loc, &C[i][j], 8, 8, 8, 1, 1);
       TRIPS_dma_wait(1);
    }
 }
}
```

The resulting code above does not overlap communication and computation: the reads synchronize before computation is allowed to proceed and the writes must complete before the next reads are allowed to execute. The software pipeline is created in later phases.

The initial context of each kernel function running on the slave processors must be transferred from the master processor. R-Stream and the thin runtime do this by packaging all the needed data in a context structure and passing a pointer to this structure to the slave processor. The kernel function first transfers the data from the remote copy of context into a local context variable (via DMA) so that it can be quickly accessed:[3]

```
typedef struct {
 double (*A)[128];
 double (*B)[128];
 double (*C)[128];
} context_t;

void kernel(int proc_id, void * arg) {
 // Global arrays
 double (*C)[128];
 double (*A)[128];
 double (*B)[128];
```

[3]This speeds dispatch vs. passing the context through shared memory.

```
 context_t context;

 // Transfer the context over.
 TRIPS_dma_get(arg, &context, sizeof(context), 0, 0, 1, 0);
 TRIPS_dma_wait(0);

 A = context.A;
 B = context.B;
 C = context.C;

 // Rest of the kernel ...
}
```

The setup code on the master processor becomes:

```
context_t context;
context.A = A;
context.B = B;
context.C = C;

TRIPS_mapped_begin(0, 4, 0, kernel, &context, sizeof(context));
```

For efficiency, R-Stream hides the latency of communication by software pipelining the DMA and computation code. First, it splits each local variable into two buffers and allocates a pointer for each buffer:

```
 double A_loc_buf1[128];
 double A_loc_buf2[128];
 double B_loc_buf1[128];
 double B_loc_buf2[128];
 double C_loc_buf1;
 double C_loc_buf2;
 double* A_loc_ptr1;
 double* A_loc_ptr2;
 double* B_loc_ptr1;
 double* B_loc_ptr2;
 double* C_loc_ptr1;
 double* C_loc_ptr2;

 A_loc_ptr1 = A_loc_buf1;
 A_loc_ptr2 = A_loc_buf2;
 C_loc_ptr1 = &C_loc_buf1;
 C_loc_ptr2 = &C_loc_buf2;
 B_loc_ptr1 = B_loc_buf1;
 B_loc_ptr2 = B_loc_buf2;
```

Next, a macro is generated to swap the buffer pointers:

```
#define TRIPS_swap() {
 double* __A_loc_tmp;
```

```
 __A_loc_tmp = A_loc_ptr1;
 A_loc_ptr1  = A_loc_ptr2;
 A_loc_ptr2  = __A_loc_tmp;
 double* __C_loc_tmp;
 __C_loc_tmp = C_loc_ptr1;
 C_loc_ptr1  = C_loc_ptr2;
 C_loc_ptr2  = __C_loc_tmp;
 double* __B_loc_tmp;
 __B_loc_tmp = B_loc_ptr1;
 B_loc_ptr1  = B_loc_ptr2;
 B_loc_ptr2  = __B_loc_tmp;
}
```

R-Stream pipelines the kernel code by performing loop shifting on the DMA operations. The resulting code prefetches the data needed for the next `j`-iteration while operating on the current one. Two DMA tags are used to distinguish between the two groups of DMA operations. Tag 0 is for read and tag 1 is for write.

```
 for (int i = proc_id; i <= 127; i += 4) {
    // Prologue
    TRIPS_dma_get(&A[i][0], &A_loc_ptr2[0], 1024, 8, 8, 1, 0);
    TRIPS_dma_get(&B[0][0], &B_loc_ptr2[0], 8, 1024, 8, 128, 0);
    TRIPS_dma_wait(0);
    TRIPS_swap();
    TRIPS_dma_get(&A[i][0], &A_loc_ptr2[0], 1024, 8, 8, 1, 0);
    TRIPS_dma_get(&B[0][1], &B_loc_ptr2[0], 8, 1024, 8, 128, 0);
    *(C_loc_ptr1) = 0;
    for (int j = 0; j <= 127; j++) {
       *(C_loc_ptr1) = *(C_loc_ptr1) + A_loc_ptr1[j] * B_loc_ptr1[j];
    }
    TRIPS_dma_put(C_loc_ptr1, &C[i][0], 8, 8, 8, 1, 1);
    // Steady state
    for (int j = 1; j <= 126; j++) {
       TRIPS_dma_wait(0); // Wait for previous get to complete.
       TRIPS_swap();
       // Prefetch
       TRIPS_dma_get(&A[i][0], &A_loc_ptr2[0], 1024, 8, 8, 1, 0);
       TRIPS_dma_get(&B[0][1 + j], &B_loc_ptr2[0], 8, 1024, 8, 128, 0);
       *(C_loc_ptr1) = 0;
       for (int k = 0; k <= 127; k++) {
          *(C_loc_ptr1) = *(C_loc_ptr1) + A_loc_ptr1[k]*B_loc_ptr1[k];
       }
       TRIPS_dma_wait(1); // Wait for previous put to complete.
       TRIPS_dma_put(C_loc_ptr1, &C[i][j], 8, 8, 8, 1, 1);
    }
    // Epilogue
    TRIPS_dma_wait(0);
    TRIPS_swap();
    *(C_loc_ptr1) = 0;
    for (int j = 0; j <= 127; j++) {
       *(C_loc_ptr1) = *(C_loc_ptr1) + A_loc_ptr1[j] * B_loc_ptr1[j];
    }
    TRIPS_dma_wait(1);
    TRIPS_dma_put(C_loc_ptr1, &C[i][127], 8, 8, 8, 1, 1);
```

```
    TRIPS_dma_wait(1);
 }
}
```

The previous transformations allocate local arrays onto the local stack of the slave processors. On TRIPS, it is better to map local arrays onto the stream register file to speed up accesses. This can be accomplished by replacing the previous buffer pointer allocation code with the following:

```
 A_loc_ptr1 = TRIPS_get_local_memory(... offset for A1 ...);
 A_loc_ptr2 = TRIPS_get_local_memory(... offset for A2 ...);
 B_loc_ptr1 = TRIPS_get_local_memory(... offset for B1 ...);
 B_loc_ptr2 = TRIPS_get_local_memory(... offset for B2 ...);
 C_loc_ptr1 = TRIPS_get_local_memory(... offset for C1 ...);
 C_loc_ptr2 = TRIPS_get_local_memory(... offset for C2 ...);
```

Finally, R-Stream must map all global data onto a shared memory segment to make them accessible to DMA. This is accomplished by replacing the initialization code running on the master processor with the following:

```
typedef struct {
 double (*A)[128];
 double (*B)[128];
 double (*C)[128];
} context_t;

typedef struct {
 double A[128][128];
 double B[128][128];
 double C[128][128];
 context_t context;
} shared_t;

int main(...) {
 // Global arrays A, B, and C and context are allocated
 // in the shared segment.
 shared_t shared = (shared_t *) TRIPS_get_shared_memory(0);
 double (*A)[128] = shared->A;
 double (*B)[128] = shared->B;
 double (*C)[128] = shared->C;

 ...

 // Create a context object.
 shared->context.A = A;
 shared->context.B = B;
 shared->context.C = C;

 // Run the kernel in parallel.
 TRIPS_mapped_end(
       TRIPS_mapped_begin(0, 4, 0, kernel, &shared->context,
                          sizeof(shared->context)));
 ...

}
```

8.2.5 Achievement and Current Status

In the project, Reservoir verified the correctness of these mapping procedures on the TRIPS simulators, and to a limited degree on the TRIPS hardware. Beyond mapping R-Stream to TRIPS, significant engineering effort was dedicated by Reservoir engineers to improve the robustness of the mapper and the infrastructure, and to explore the interaction of the mapper with the TRIPS compiler.

The R-Stream compiler is available to support TRIPS applications and transitions, and for other purposes. For government purposes, the source code is available on a limited rights basis. Academic and industry license are also available. Additional information about licenses and usage can be obtained from Reservoir Labs.

While the underlying mathematics for polyhedral program optimization has existed for more than 20 years, R-Stream represents an advance in the art of high-level optimization for many reasons, including:

1. R-Stream builds on and improves relatively recent algorithms [48, 4] for lowering code (so called "polyhedral scanning") from the polyhedral representation. Paradoxically, prior to these algorithms, it was not possible to efficiently generate code that had been optimized through polyhedra. Reservoir's innovations shape the generation of code from these models to allow LLCs to generate machine code from the parallelized code.
2. R-Stream provides an end-to-end mapping process with several new algorithms that address specific aspects of polymorphous computer architectures, such as explicitly controlled communications and the the management of fine-grained local scratch pad memories.
3. R-Stream provides implementations of polyhedral parallelization algorithms that previously had been published in the literature, but not actually implemented. Because those authors had generated their examples by hand, there were significant gaps ("and then a miracle occurs") in the published algorithms. R-Stream provides a working implementation of these algorithms and thus fills these gaps. R-Stream also provides scalable implementations of these algorithms.
4. R-Stream provides a mapper that proceeds entirely in the polyhedral domain. This enables the development of "fused" optimization phases for finer trade offs between competing concerns in mapping steps.
5. R-Stream provides algorithms and implementations for mapping that extend the scale of programs that can be mapped.
6. R-Stream implements a polyhedral mapper on top of a robust scalar compiler infrastructure that allows abstracting the result from syntactic considerations and provides for a source-to-source mapping process that works compatibly with the quirky requirements of low-level compilers.

Any one of the above innovations distinguishes R-Stream from other high-level optimizers. R-Stream provides them all in one package. DARPA sponsorship of R-Stream has taken a theoretically plausible advanced capability, and solved the difficult engineering and research challenges to a successful demonstration.

As of this date (Fall 2008), the R-Stream compiler is being actively improved in follow-on research projects with the Department of Energy and the Department of Defense. The capabilities in R-Stream developed with DARPA TRIPS funding contribute to the generality and robustness of the mapping algorithms. Subsequent research and development activities at Reservoir have expanded the scope of applicability beyond the static control model to include data-dependent predication. New target processors and computational accelerators are being supported, including FPGA, GPU, tiled arrays, and wide SIMD engines. R-Stream can also optimize code for symmetric multiprocessors, emitting code with OpenMP annotations. The compiler is being developed to support missions in advanced signal processing, cyber security, and scientific computing. New parallelization algorithms that trade locality with parallelization have been developed, as well as fused tiling-placement-communication generation phases.

8.2.6 High-Level Optimization for Polymorphous Computing Architectures

R-Stream supports the vision of polymorphous computing by providing a formalization and implementation of the meaning of "mapping" to Polymorphous Computing Architectures. Achieving this result has not been easy: our experience in early iterations in the PCA Morphware Forum to define programming models, machine models and streaming/threaded virtual machines suffered from the lack of the item that connects them–the mapping algorithm. Now, with a mapping algorithm in hand, at the completion of the PCA program, it is possible to sensibly ask and answer questions on these issues.

For example, in the development of a new PCA programming model, a mapper allows identifying features that duplicate compiler capabilities or interfere with them, versus features that assist the mapper. One initial conclusion is that streaming programming models often interfere with the mapping process. We found it easier in the end to move away from streaming programming models to a more declarative, abstractable form that can be expressed using standard C.[4]

Similarly, for a PCA/Morphware machine model, we can now identify features that are suited to the mapping optimization process. A machine model feature that is disconnected from the mapping process is merely prose that cannot be used for automated optimization. Finally, we found that making progress on the definition of PCA streaming virtual machines (SVM) was impossible before a consistent definition of mapping to specific execution models was defined. We stopped pushing the SVM development path in order to focus on simpler execution models for which mapping was well-defined. With this mapper now in hand, it would be a good time to work on a new SVM abstraction.

Work in PCA and the Morphware Forum included discussion of dynamic responses, e.g., reconfiguring architectures and remapping applications in the face of changing resources or application demands. R-Stream is not a dynamic compiler (yet), but it provides a new reference for the complexity of this process that should assist in the understanding of the feasibility of dynamic mapping. There is tension: the very hardware features in PCA architectures that provide for high performance and power efficiency, which are in essence the explicit management of hardware, make the automatic mapping problem complex. R-Stream demonstrates how programs can be mapped, automatically and statically. Making this happen dynamically, without compromising the efficiency of the PCA hardware architectures, is the next challenge.

For dynamic architecture reconfiguration, the mapper is probably further away. While we did some early work to try to formalize the notion of dynamic changes to the target architecture, by making the machine model a first class object, more needs to be done. Formalizing dynamic hardware changes is well outside any parallel optimization framework we know, including the polyhedral model.

8.2.7 Future Research Opportunities

Beyond PCA, R-Stream represents an ideal platform for the development of new compilation capabilities. In the case of adaptive compiler techniques (iterative compilation, continuous compilation) the results that have been published to date in the literature have generally been limited to modest performance improvements for scalar optimization, through adjustment of mapping cost functions and various refinements of the time honored strategy of "pushing all the buttons", i.e., Python scripts fiddling with compiler command line switches. The application of these adaptive compiler techniques to the parallel computing domain will be thwarted without a robust framework for exploring the vast space of parallel mapping options. Early work shows that the polyhedral representation provides the potential to bypass fundamentally futile searches of vast mapping spaces by providing explicit framings of highly-fused parallelization optimizations as linear programs [47, 46]. The dimensionality of adaptive choices can be reduced similarly to the tuning of cost parameters to allow adaptive compilation to feasibly proceed.

The polyhedral mapper also highlights the limitations of dependence-preserving transformations and the need for very high level transformations that explore algorithm variants. In an expression-level exploration that is beyond simple

[4]Note that this is in contrast to the steaming *execution model*. The streaming execution model - with explicit control of the communications and memory is considered to be essential to achieving high performance and efficiency. R-Stream provides a way to bridge from a more abstract general programming model to the streaming execution model.

polyalgorithm approaches, a coupled polyhedral mapper may provide more aggressive pruning and guidance, or even jointly-formulated explicit exploration with polyhedral mapping, within certain application domains. To do this in a sound manner requires programming language and compiler technologies to be developed that allow mapping to be made within formalizations of the algorithm objectives, e.g., formalization of numerical properties. This extends our general observation that the most productive programming language route for enabling new automatic optimization capabilities is through the definition of more abstract, declarative and higher-level programming abstractions.

Polyhedral mapping also opens the door to research in computer architecture mechanisms at the high level. With a powerful mapper, it is possible to evaluate different high-level architecture tradeoffs, such as mechanisms for massively multicore chips, or massive Exascale systems. The mapper reduces the obstacle of programming complexity in deploying and using new architecture features that provide for high performance and efficiency.

9 Evaluation of TRIPS System

This section describes a detailed performance analysis that explores how well the TRIPS compiler and hardware meets its goals of exploiting concurrency, hiding latency, and distributing control. Using the TRIPS hardware and detailed microarchitectural simulators to gather detailed statistics not available from the hardware, we compare the EDGE ISA, microarchitecture, and performance to modern processors using hand-optimized and compiled benchmarks. We find that the EDGE ISA incurs a substantial overhead in total number of instructions fetched and executed, relative to conventional RISC architectures like the PowerPC, because of extensive predication and instruction overheads required by the dataflow model.

Our microarchitecture analysis shows that TRIPS can keep much of the instruction window full; compiled code shows an average of 400 total instructions in flight (887 peak for the best benchmark) and hand-optimized code shows an average of 630 (1013 peak). While much higher than conventional processors, the number of instructions in flight is less than the maximum of 1024 because the compiler does not completely fill blocks and the hardware experiences pipeline stalls and flushes due to I-cache misses, branch mispredictions, and load dependence mispredictions. A strength of the EDGE ISA and distributed control is that TRIPS requires less than half as many register and memory accesses as the PowerPC because it converts these into direct producer/consumer communications[5]. Furthermore, the communicating instructions are usually on the same tile or an adjacent tile, which makes them power efficient and minimizes latency.

We compare the performance of TRIPS to the Intel Core 2, Pentium III, and Pentium 4 using hardware performance counters on compiled and hand-optimized programs. To normalize for different process technologies and design styles (ASIC versus custom), we use processor clock cycles as the metric of performance. We find that TRIPS outperforms the Core 2 by an average factor of 3 on hand-optimized benchmarks and that the Core 2 outperforms the Pentium 3 and Pentium 4 by an additional factor of 2. These results demonstrate the performance potential for the TRIPS processor, assuming highly optimized code. On the EEMBC suite, the Core 2 outperforms (in cycles executed) TRIPS compiled code by 30%. On SPEC-2000, TRIPS compiled code achieves only half the performance of the Core 2 on integer benchmarks but matches the performance of the Core 2 on floating point benchmarks.

These experiments suggest that TRIPS-like processors have the capability to achieve substantial performance improvements over conventional microprocessors by exploiting concurrency. However, realizing this performance potential relies on the compiler to better expose concurrency and create large blocks of TRIPS instructions, as well as microarchitectural innovations in control distribution and branch prediction.

9.1 Evaluation Methodology

We evaluate the EDGE ISA and its TRIPS microarchitecture and compare its performance with conventional architectures using the on-board performance counters in the TRIPS hardware and in commercial platforms. We also use TRIPS simulators and a PowerPC simulator to gain deeper insights in Sections 9.2 and 9.3. All performance measurements in Section 9.4 are from the actual hardware.

TRIPS Prototype System: A TRIPS chip consists of two processors that share a 1 MB L2 static NUCA [32] cache and 2 GB of DDR Memory, but we use only one processor in all experiments. Each processor has a private 32 KB L1 data cache and a private 80 KB L1 instruction cache. The processor and memory speeds can be adjusted using a phased-lock loop (PLL); all of the experiments run the processor core at 366 MHz and the DRAM with 100/200 MHz DDR clocks. TRIPS system calls interrupt program execution, halt the processor, and are proxied to an off-chip commercial processor running Linux. Because the TRIPS cycle counters increment only when the processor is not halted, the program performance measurements ignore the time to process system calls. The tools we use to measure cycles in the commercial systems also exclude operating system execution time, thus providing a fair comparison.

[5] We use the PowerPC ISA as the baseline to represent modern RISC processors. The x86 ISA is a poor comparison for instruction efficiency because of its CISC instructions and few registers.

Table 8: Evaluation Reference Platforms

System	Technology	Processor Speed	Memory Speed	Proc/Mem Speed Ratio	L1 Cache Capacity (D/I)	L2 Cache Capacity	Memory Capacity
TRIPS	130nm	366 MHz	200 MHz	1.83	32 KB / 80 KB	1 MB	2 GB
Core 2	65nm	1600 MHz	800 MHz	2.00	32 KB / 32 KB	2 MB	2 GB
Pentium 4	90nm	3600 MHz	533 MHz	6.75	16 KB / 150 KB	2 MB	2 GB
Pentium III	250nm	450 MHz	100 MHz	4.50	16 KB / 16 KB	512 KB	256 MB

Table 9: TRIPS Benchmark Suites

Suite	# of Benchmarks	Characteristics
Kernels	4	transpose (ct), convolution (conv), vector-add (vadd), matrix multiply (matrix)
VersaBench	3 of 10	bit and stream (fmradio, 802.11a, 8b10b)
EEMBC	28 of 30	Embedded benchmarks
Simple	15	Hand-optimized versions of Kernels, VersaBench, and 8 EEMBC benchmarks
SPEC 2000 Int	9 of 12	All but gap, vortex and C++ benchmarks
SPEC 2000 FP	9 of 14	All but sixtrack, and 4 Fortran 90 benchmarks

Simulators: We use a functional TRIPS simulator and a low-level microarchitecture simulator to gather statistics not available from the hardware [71]. A Power PC functional simulator [53] produces statistics that measure loads, stores, and register accesses from gcc compiled PowerPC–AIX binaries.

Reference Platforms: We compare performance of the TRIPS prototype to three reference platforms from the Intel x86 product family (Pentium III, Pentium 4, and Core 2) using the PAPI software package to access the Intel processors' hardware counters [45]. Because each machine is implemented in a different process technology, we use cycle counts as the relative performance measure. Table 8 shows the platform configurations including processor and DDR DRAM clock speed and the memory hierarchy capacities. Cycle count is an imperfect metric because some architectures, particularly the Pentium 4, emphasize clock rate over cycle count. However, we expect that the TRIPS microarchitecture, with its partitioned design and no global wires, could be implemented in a clock rate equivalent to or faster than the Core 2, given a custom design and the same process technology. Another pitfall with this comparison is that the relatively slow clock rate of TRIPS may make memory accesses less expensive relative to high clock-rate processors. To account for this, we under-clocked the Core 2 from 1.8 GHz to 1.6 GHz to equalize the processor/memory speed ratio to TRIPS. However, a slower clock has little effect on the measured applications because they are largely L2 cache resident.

Benchmarks: Table 9 shows the benchmark suites, from simple kernels to complex uniprocessor workloads. We compiled these applications with the TRIPS C and Fortran compiler [60], and hand-optimized the compiler-generated TRIPS assembly code to study the performance potential of TRIPS. We extensively hand-optimized four scientific kernels on TRIPS: matrix transpose (ct), convolution (conv), vector add (vadd), and matrix multiply (matrix). In addition, we hand-scheduled matrix and vadd to achieve high performance. We hand-optimized three stream and bit operation benchmarks from the VersaBench suite [49] and eight medium-sized benchmarks from the EEMBC suite [15]. The most complex benchmarks come from SPEC2000 and include nine integer and nine floating-point benchmarks [63]. SimPoint regions are used to select appropriate simulation points for a detailed evaluation of the SPEC benchmarks [59]. Section 9.6 details the performance and optimization of a synthetic aperture radar (SAR) application on TRIPS.

9.2 ISA Evaluation

This section examines how well the compiled and hand-optimized programs map into the ISA and characterizes block size, instruction overheads, and code size. We compare compiled and hand-optimized TRIPS programs to programs compiled for a RISC ISA (PowerPC) processor to quantify the relative overheads of the EDGE ISA. We also present the

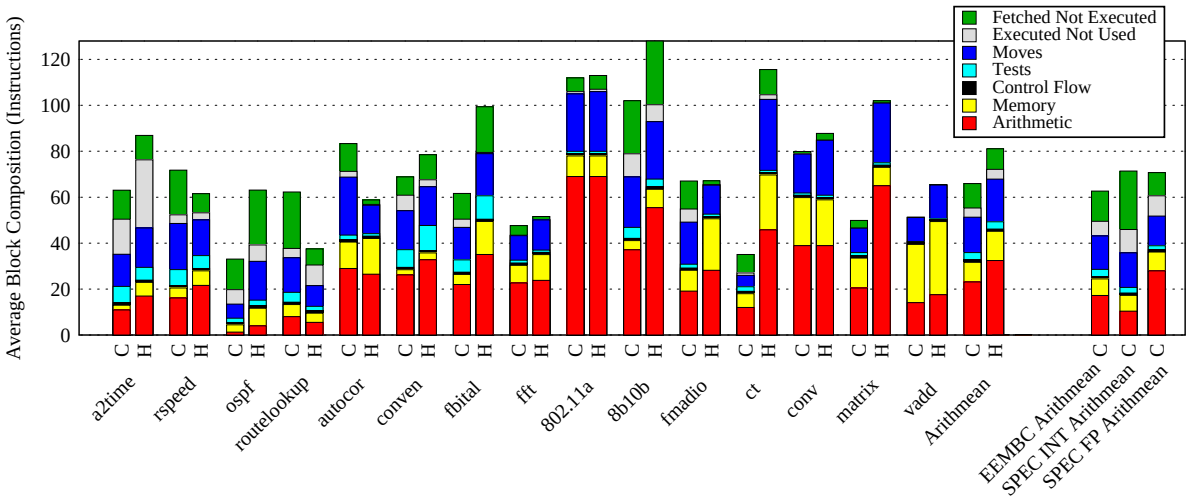


Figure 21: TRIPS Block Size and Composition for Compiled (C) and Hand-optimized (H) Benchmarks

means of the EEMBC, SPEC INT, and SPEC FP suites to show the impact of the application characteristics. All of the data in this section was gathered through simulation because of the nature of the statistics needed.

9.2.1 TRIPS Block Size and Composition

A key parameter in specifying a block-atomic EDGE ISA is the block size. Early experience demonstrated that creating programs with average block sizes of 20+ instructions was not difficult with standard compiler transformations, and that larger blocks would yield larger effective instruction window, better amortize block overheads, and have the potential for better performance. We chose a maximum block size of 128 instructions as an aggressive compiler target, to stress the software system to form larger blocks.

Figure 21 weighs each block's size by execution frequency to show the average block size, and breaks out the number of arithmetic instructions, memory instructions, branch/jump/call/return instructions, test instructions (used for branches and predication), and `move` instructions (used to fan out intermediate operands). The figure does not include the register read/write instructions since they reside in the block header and are not part of the 128 instructions. Fetched Not Executed instructions were fetched speculatively but never executed, either because they did not receive a matching predicate, or because they did not receive all of their operands due to predicated instructions earlier in the block's dataflow graph. Executed Not Used instructions were fetched and executed speculatively but their values were not used due to predication later in the dependence graph.

For some programs, such as a2time, the number of mispredicated instructions accounts for half the total instructions within a block. A2time contains several nested `if/then/else` statements; to minimize the number of blocks executed, the compiler produces code that speculatively executes both the `then` and `else` clauses simultaneously within one block and inserts a predicate computation to select the correct outputs. Nonetheless, this aggressive predication can improve system performance because it eliminate branch mispredictions and enables the implementation of a pipeline with higher front-end fetch bandwidth.

The remainder of the instruction types, tests, control flow, memory, and arithmetic, are required for correct execution. The number of useful instructions (excluding `move` and mispredicated instructions) varies. Some programs with complex control have only ten instructions per block while others with more regular control have as many as 80 instructions per block. To implement dataflow execution in a block, the EDGE ISA requires `move` instructions. First, since a TRIPS instruction has a fixed width, it can target at most two consumers. The compiler must therefore insert `move` instructions to fanout values consumed by more than two instructions. Second, predicate merge points, corresponding to *phi* merge nodes in the dataflow graph, sometimes require predicated `move` instructions. The result is that `move` instructions account for nearly 20% of all instructions in a block, more than anticipated at the start of the design.

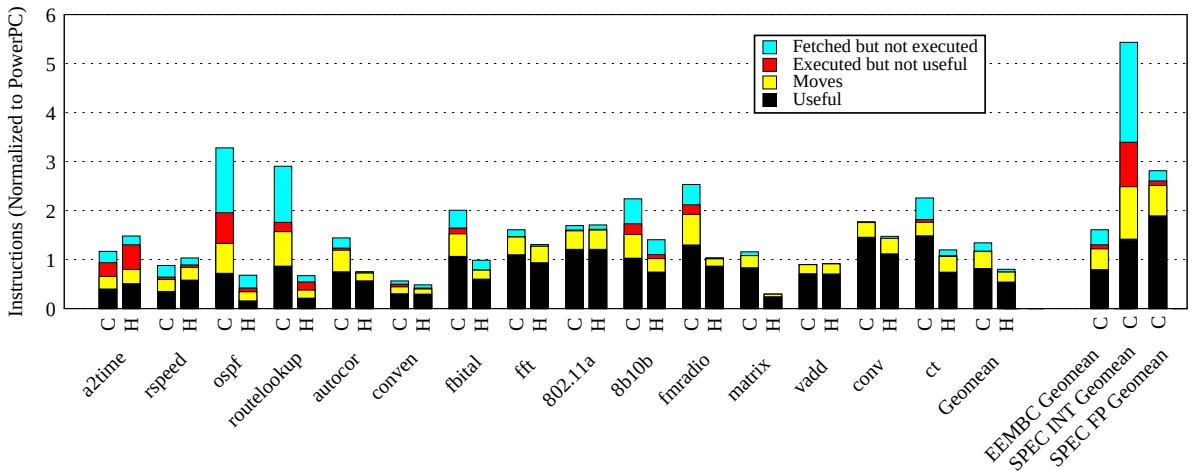


Figure 22: TRIPS Instructions Normalized to PowerPC for Compiled (C) and Hand-optimized (H) Benchmarks

Compiled code has an average block size of 64 instructions, but with high variance, ranging from 30 to over 110 instructions. Hand-optimizations to improve performance further increase block size. For example, the hand-optimized versions of ospf and ct have blocks two and three times larger than their respective compiled versions. These increases are often driven by instruction optimizations that decrease block size followed by opportunities to merge adjacent smaller blocks or by increasing unrolling factors to fill blocks. In summary, both the hand-optimized and compiled code utilize the aggressive 128-instruction block size to achieve average block sizes ranging from 20 to 128. To expose concurrency and aggressively speculate, the ISA overheads include `move` instructions and useless instructions, which are a significant fraction of the in-flight instructions.

9.2.2 TRIPS ISA versus PowerPC

To quantify the differences between an EDGE ISA and a popular RISC ISA, we compare to the PowerPC. Figure 22 shows fetched instruction counts on TRIPS normalized to PowerPC. Because of limitations in the current PowerPC infrastructure, the following graphs includes the mean of only 8 of the 18 SPEC benchmarks (5 INT and 3 FP). To compare expressive power of the compute instructions, the TRIPS instructions do not include register read/write instructions that appear in the block header, nor `NOP`s in underfull blocks. For both TRIPS and PowerPC, the instruction count omits incorrectly fetched instructions due to branch mispredictions.

Not surprisingly, the number of useful instructions executed on TRIPS and PowerPC are similar because the TRIPS ISA is composed of RISC-style instructions. On compiled code, TRIPS tends to execute more instructions due to prototype simplifications, which introduce inefficiencies in constant generation and sign extension unrelated to its execution model. For hand-optimized benchmarks, TRIPS executes fewer instructions because its larger register set (128 registers) eliminates store/load pairs and because more aggressive unrolling exposes more opportunities for instruction reduction. The number of fetched but mispredicated instructions varies across the benchmarks, depending on the degree of predication. Overall, TRIPS may need to fetch as many as 2–6 times more instructions than the PowerPC, due to aggressive predication.

9.2.3 Register and Memory Access

TRIPS inter-block communication is through registers and memory while intra-block communication is direct between instructions, reducing the number of accesses to registers and memory. The TRIPS prototype has a total of 128 registers spanning four register banks (32 registers per bank); each bank has only one read and one write port. The larger register file benefits the memory system as fewer register fills and spills are required. Eliminating store/load pairs ultimately

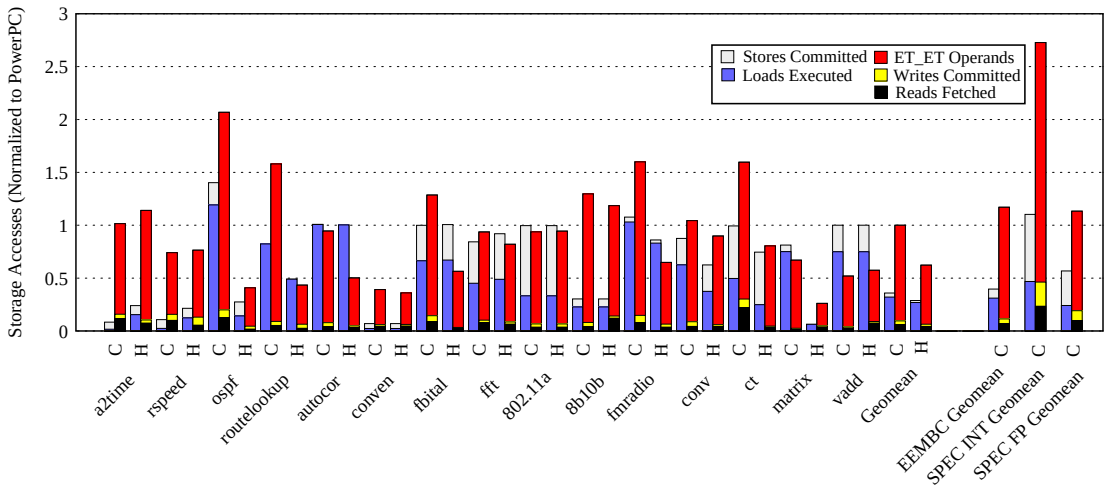


Figure 23: Storage Accesses Normalized to PowerPC for Compiled (C) and Hand-optimized (H) Benchmarks

improves performance as communication through registers is faster than communication through memory [40]. Compared to a conventional architecture, TRIPS replaces memory instructions for less expensive register reads and writes, and replaces register reads and writes for less expensive direct communication between producing and consuming instructions.

The left bar stack of each pair in Figure 23 shows the number of loads and stores on TRIPS normalized to the total number of loads and stores on the PowerPC. On average, TRIPS executes about half as many memory instructions as the PowerPC and as few as 15%, due to the bigger register file and direct communication. Several of the hand-optimized benchmarks have significantly fewer memory accesses than the compiled versions because they register allocate fields in structures and small arrays, whereas the compiler currently does not register allocate these. The right bar stack shows the number of register file reads, writes, and operand network communications on TRIPS normalized to the total number of register file reads and writes on the PowerPC. Because of direct operand communication, TRIPS requires only 10–20% of the register accesses needed on the PowerPC. The top bar of the stack shows that the relative number of operands transmitted through direct communication far exceeds the number of register reads and writes on TRIPS.

Comparing each hand-optimized benchmark to its compiled counterpart on average, there are fewer register accesses, OPN communications, and memory accesses. The hand-optimized version eliminates many memory accesses by exploiting programmer knowledge about pointer aliasing to aggressively allocate more variables into registers. It also has fewer instructions because it removes instructions with aggressive peephole optimizations and eliminates unnecessary sign extensions. On average, the sum of register reads, writes, and direct communication is about the same as the number of PowerPC register reads and writes. On some benchmarks, specifically SPEC INT, the temporary communication is large because of the distribution of predicates and communication of useless values by mispredicated instructions. Figure 22 shows that the SPEC INT benchmarks fetch approximately half useless instructions, which leads to much more communication than on the PowerPC. In a conventional architecture, the register file broadcasts an instruction's result to other instructions. In TRIPS, fanout may require a tree of `move` instructions, which increases the number of copies of the original operand that are communicated.

9.2.4 Code Size

The TRIPS ISA increases dynamic code size over a PowerPC significantly. Each block has 128 32-bit instructions, a 128-bit header, 32 22-bit read instructions, and 32 six-bit write instructions. The compiler inserts `NOP`s when a block has fewer than 32 reads or writes or fewer than 128 instructions. `NOP`s consume space in the L1 I-cache but are not executed. We compared the dynamic code of TRIPS to the PowerPC by computing the number of unique instructions that are fetched during execution. The dynamic code size of TRIPS averages about 6 times larger than the PowerPC, but with a wide variance. The number of unique useful instructions for TRIPS is about two and three times that of the PowerPC,

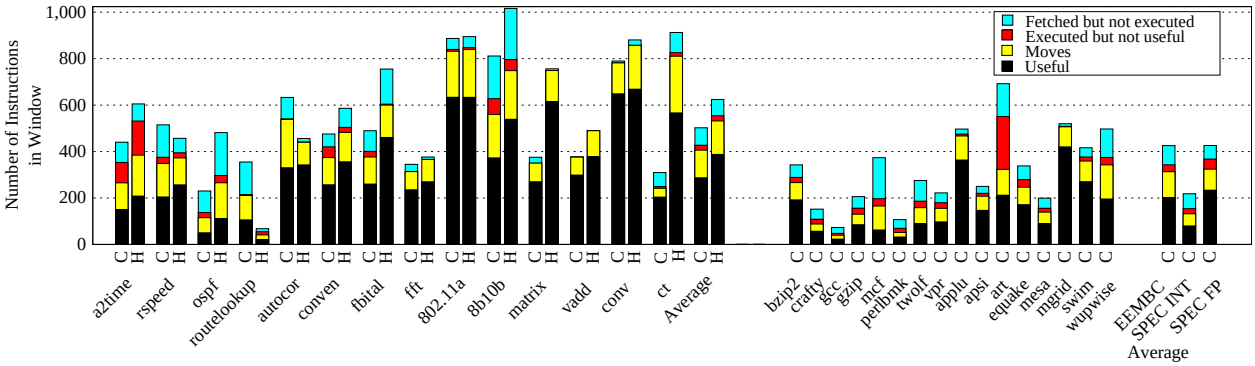


Figure 24: Average Number of in flight Instructions for Compiled (C) and Hand-optimized (H) Benchmarks

indicating that instruction replication due to TRIPS block optimizations accounts for about half of the code bloat. The `move` instructions and the block header (including useful register read and write instructions) account for about 30% of the total instructions.

The TRIPS prototype compresses underfull instruction blocks in memory and in the L2 cache down to 32, 64, or 96 instructions, depending on block capacity, which reduces the expansion factor over PowerPC to a factor of 4. Block compression in the instruction cache may unduly slow down instruction fetch or require more complex instruction routing from the instruction cache banks to the execution tiles. Experiments generally show a low instruction cache miss rate on small and medium sized benchmarks, but some SPEC benchmarks have miss rates in the range of 15–25%, indicating that instruction cache pressure is a serious problem for real applications. Fortunately, partitioned architectures, such as TRIPS, that bank the instruction cache can be easily designed with larger overall instruction caches to mitigate this type of cache pressure.

9.3 Microarchitecture Evaluation

The primary goal of the TRIPS microarchitecture is to support a large instruction window with a partitioned design. One important aspect is the fraction of the instruction window that is full, which depends on the TRIPS block predictor. Another aspect is the bandwidth of the partitioned memory system and the usage of the operand network. This section explores these unique aspects using detailed statistics gathered from simulation.

9.3.1 Filling a 1K Instruction Window

Each TRIPS block contains up to 128 instructions and the hardware can execute up to eight blocks concurrently so the maximum dynamic instruction window size, with full blocks and accurate speculation size, is 1024 instructions. Figure 24 shows the average number of TRIPS instructions in the window across a variety of hand and compiled benchmarks. This metric multiplies the average number of blocks in flight (speculative and non-speculative) and the dynamic average number of instructions per block. Compiled codes, produce an average of 400 total instructions of which 170 are useful. The hand-optimized programs with larger blocks achieve a mean of 630 total instructions, more than 380 of which are useful. Compared with issue windows of 64 and 80 on modern superscalar processors, TRIPS exposes more concurrency, but at the cost of more communication.

In addition to predication, the principal speculation mechanisms in TRIPS are the store-load dependence predictor and the next-block predictor. When the load/store queue detects that a speculatively issued load has executed incorrectly, it flushes the block pipeline and enters the load into the dependence predictor's simple partitioned load-wait table in the data tile. For the SPEC benchmarks, the predictor is effective in part because the compiler reduces the number of loads and

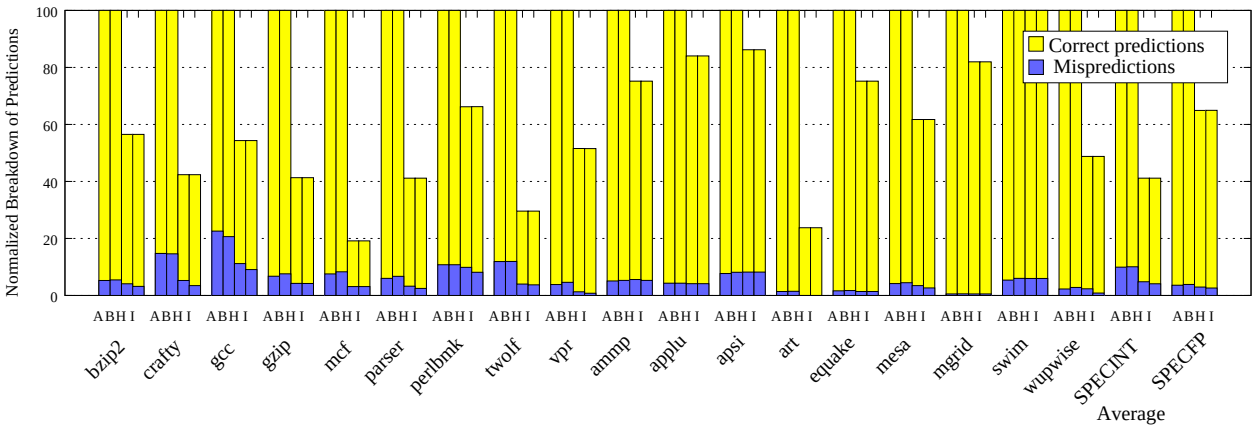


Figure 25: TRIPS Block Prediction Accuracy

stores (as discussed in Section 9.2.3), resulting in fewer than one block flush per 2000 useful instructions, without overly constraining speculative load issue.

The TRIPS next-block predictor selects the next block to be fetched [55]. It consists of a 5 KB local/global tournament exit predictor that predicts which exit branch will be taken from the TRIPS block (one of up to eight) and a 5 KB multi-component target predictor that predicts the target address of this exit branch. Figure 25 shows the correct/misprediction breakdown for four different configurations: the first bar (A) shows the breakdown for an Alpha 21264-like conventional tournament branch predictor predicting TRIPS-compiled basic block code, the second bar (B) shows the TRIPS block predictor predicting basic block code, the third bar (H) shows the TRIPS prototype block predictor predicting hyperblock code, and the final bar (I) shows a "lessons learned" TRIPS block predictor that could be used in future designs constructed by scaling up the target predictor component to 9 KB. Each bar is normalized to the total number of predictions made for basic block code. The average MPKI (Mispredictions Per 1000 Instructions, omitting move and mispredicated instructions) observed for these four configurations on SPEC INT are 14.9, 15.1, 8.6 and 7.3 respectively. SPEC FP applications have an MPKI of 1.6, 1.7, 1.5 and 1.3 respectively.

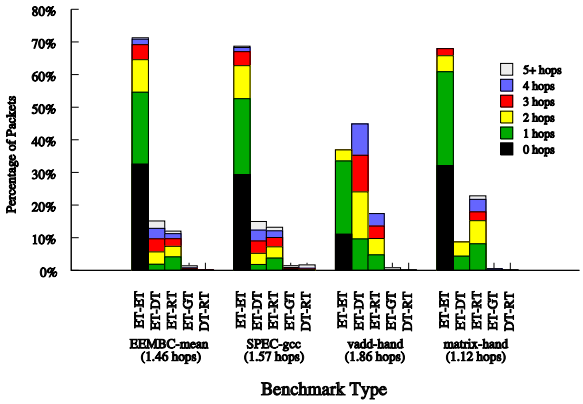
Predicting TRIPS blocks rather than basic blocks can improve accuracy, because hard-to-predict branches are converted to predicates, and can degrade accuracy, because predication can obscure correlated branches in the history. Although the prototype predictor (H) has a higher misprediction rate than a conventional predictor (A), it has a lower MPKI because it makes fewer predictions (59% fewer on SPEC INT and 35% fewer on SPEC FP). The improved TRIPS predictor I, reduces SPEC INT MPKI by 16% and SPEC FP MPKI by 14%. More sophisticated multi-component long-history predictors [28, 58] could be used to improve the TRIPS predictor, as well as improving the efficiency of the exit encoding. Additionally, increasing the size of the currently small branch target buffer, call target buffer, and history will improve accuracy. Lower prediction accuracy has a significant effect on the utilization of the instruction window and, as discussed in Section 9.4, has a strong correlation with performance. More aggressive next-block predictors would dramatically improve prediction accuracy, but may still fall short of modern branch prediction accuracies.

9.3.2 Feeds and Speeds

In this section, we explore the performance of the banked memory system and operand network, two defining features of the distributed TRIPS implementation. For the memory system, we developed kernels that saturate the bandwidth of each of the banks, providing insight into the types of optimizations required for memory-bound programs. For the operand network, we measure traffic load to determine how well the compiler's placement algorithm minimizes the distance between communicating instructions.

	L1 D-Cache to Processor	L2 to L1	Memory to L2
Peak Ops/ cycle	4 - 8byte requests	3.2 - 16 byte requests	1 - 64 byte request
Peak BW (GBytes/sec)	10.9	17.5	5.6
Achieved BW (GBytes/sec)	10.5	17.2	3.2
% of Peak	96.5%	98.5%	57.8%

(a) Memory Bandwidth



(b) Operand Network

Figure 26: TRIPS Bandwidths at 366MHz and Operand Network (OPN) Profile with Average Hops per Packet

Memory System: The TRIPS prototype employs an address-partitioned memory system that divides the L1 data cache into four 8-KB, single-ported data banks and the L2 cache into sixteen 64-KB, single-ported memory banks. The table in Figure 26 shows the achieved memory bandwidth on the hardware at a core speed of 366 MHz for a hand-optimized vector add (vadd) kernel. With careful instruction scheduling, vadd can attain nearly 100% of the core's peak of four memory operations per cycle (10.5 GB/sec), indicating effective use of the partitioned L1 data cache. By adjusting the vector size of vadd, we constructed a microbenchmark with an access pattern to maximize the consumption of the L2 cache and main memory bandwidth. This program nearly reached the theoretical peak of the L2 and a majority of the main memory bandwidth provided by the dual DDR memory controllers. While the benchmark achieves only 57.8% of the maximum interface bandwidth, the vast majority of the loss is due to the memory controller protocol and not to the TRIPS design itself. Similar techniques and principles were used to hand-optimize dense matrix kernels [13] and lessons learned from these case studies were used to improve the compiler's instruction placement algorithms.

Operand Network: The Operand Network connects the TRIPS processor tiles and transmits operands between execution tiles, the register file tiles, and the data cache tiles [24]. The TRIPS compiler's instruction placer takes as input the tile topology and the dependencies between the instructions in each block. It optimizes the instruction placement to exploit concurrency and minimize the distance between dependent instructions along the program's critical path. The graph in Figure 26 displays the breakdown of the hop count for OPN traffic. On average, ET–ET operand traffic dominates the OPN and about half of the operands are bypassed locally within an ET resulting in an average operand hop count of 0.9. While an ideal instruction placement would use local bypassing for all operand communication (0 hops), the inherent tradeoff between locality and concurrency combined with limited instruction storage per tile demands that many communicating instructions reside on different tiles. The ET–DT and ET–RT traffic typically requires more hops (and thus longer level-1 cache latency) because the DTs and RTs lie along the edge of the ET array. For example, vadd streams data from its L1 caches, yielding high ET–DT traffic, while *matrix* primarily uses data in its register file, yielding greater ET–RT traffic. These two microbenchmarks illustrate how the OPN supports highly divergent traffic patterns. We show the results of one SPEC benchmark, *gcc*, and the mean of the EEMBC benchmarks to demonstrate that the load on the OPN is similar between these suites.

9.3.3 ILP Evaluation

The TRIPS prototype can execute up to 16 instructions per cycle (IPC), but can only sustain 16 IPC under ideal conditions: 8 blocks full of executed instructions, perfect next-block prediction, and no instruction stalls due to long-latency instructions. Actual IPC on the hardware is limited to 1/8 of the block size. Since the average block size of our hand-optimized benchmarks is 80 instructions, we could theoretically achieve at most an average IPC of 10 on them. Figure 27 shows the sustained IPC that TRIPS achieves across the benchmarks. While some applications are intrinsically

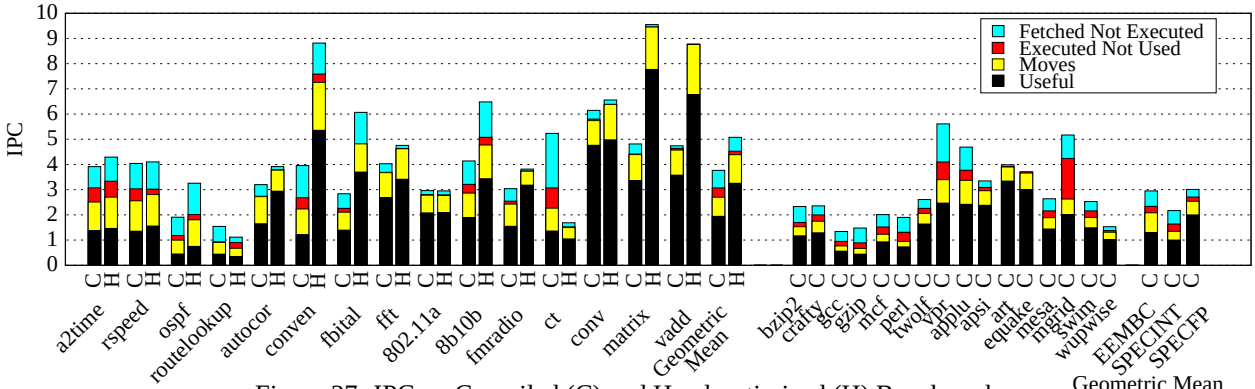


Figure 27: IPC on Compiled (C) and Hand-optimized (H) Benchmarks

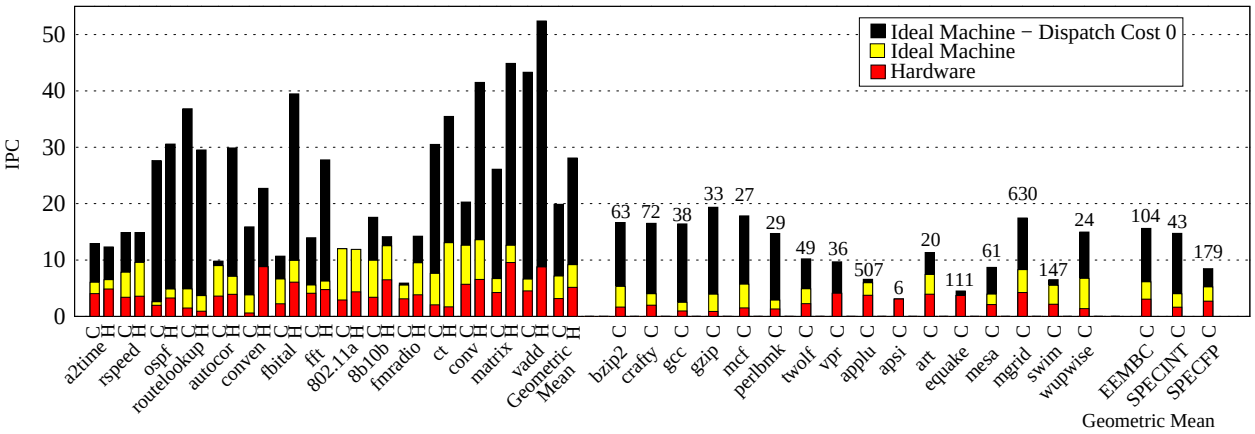


Figure 28: IPC for TRIPS and an Ideal EDGE Machine

serial (e.g., routelookup, which traverses a tree data structure serially), others reach 6 to 10 IPC, showing that the processor can take advantage of the greater ILP of these programs. The hand codes have an IPC 25% greater on average than their compiled counterparts, mostly due to better block optimization. The SPEC benchmarks have lower IPC, both because they have smaller average block sizes, and more flushes due to branch mispredictions and i-cache misses.

To understand the theoretical ILP capability of EDGE architectures, we conducted a limit study using an idealized EDGE machine with perfect prediction, perfect predication, perfect caches, infinite execution resources, and a zero-cycle delay between tiles. It, like TRIPS, has a window size of 1K instructions and a dispatch and fetch cost that only allows a new block to be started once every eight cycles. Figure 28 shows that on average this ideal machine only outperforms the prototype by roughly a factor of 2.5, indicating only moderate room for improvement due to low inherent application ILP, the dispatch cost, and limited window size. Simulating this ideal machine with a zero-cycle dispatch cost increases the IPC on average by a factor of four. However, eliminating only the dispatch delay on TRIPS improves performance by only 10%, which indicates that dispatch is not the primary bottleneck on the hardware. We also annotate the top of the SPEC bars with the IPC for the ideal machine with a window of 128K instructions and a dispatch cost of zero cycles. The SPEC benchmarks have a wide range of available ILP, with most benchmarks around 50 IPC but some FP benchmarks having IPCs in the hundreds. The simple benchmarks have a similar range of IPCs with several such as 802.11a and 8b10b that are inherently serial and do not exceed 15; others such as vadd and fmradio are concurrent but are resource limited on the hardware resulting in IPCs of 1000 and 500 respectively on the ideal machine with a 128K window. This study reveals that while the hardware has room to improve, the amount of ILP currently available to TRIPS is limited and that larger window machines have the potential to further exploit ILP.

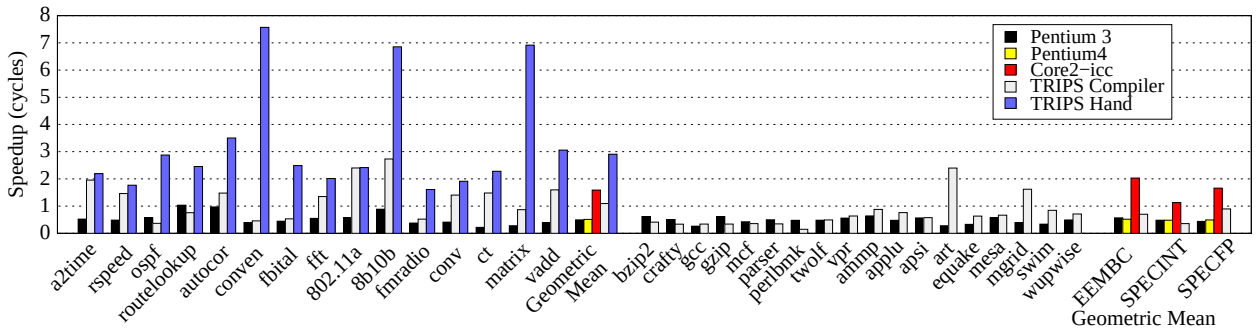


Figure 29: Speedup Relative to Core 2-gcc

9.4 Application Performance Evaluation

This section compares TRIPS to conventional processors using hand-optimized benchmarks to show the potential of TRIPS and compiled benchmarks to show the current state of the compiler. TRIPS code is compared to benchmarks compiled with both the GNU C compiler (gcc) as well as the native compiler (icc) on the reference machines. Data in this section is obtained from hardware performance counters and SPEC results are for full applications rather than the SimPoint regions used in the previous sections.

9.4.1 Simple Benchmarks

The left half of Figure 29 shows cycle counts for TRIPS hand-optimized code, TRIPS compiled code, icc-compiled code for the Intel Core 2, and gcc-compiled code for the Intel Core 2, Pentium 4, and Pentium III, normalized to the Core 2 using gcc on the simple benchmarks. The TRIPS compiler achieves equivalent performance to the Core 2 on average, with better performance on nine benchmarks and worse performance on six. Benchmarks with smaller speedups like rspeed are sequential algorithms that do not benefit from increased execution bandwidth or deep speculation. The benchmarks that show the largest speedups, such as matrix and 8b10b, typically have substantial parallelism exposed by the large window on TRIPS. The TRIPS hand-assembled code always outperforms the Core 2, with an average 2.9x speedup.

The performance differences between TRIPS compiled code and TRIPS hand-assembled code are primarily due to more aggressive block formation, unrolling, and scalar replacement. For example, 8b10b benefits from unrolling the innermost loop of the kernel to create a full 128-instruction block and from register allocating a small lookup table. In fmradio, the hand-optimized code fuses loops that operate on the same vector, and uses profile information to exclude infrequently taken paths through the kernel.

To completely remove the influence of the compiler and show the ability of TRIPS to exploit a large number of functional units, we compare a TRIPS hand-optimized matrix multiply [13] to the state-of-the-art hand-optimized assembly versions of GotoBLAS Streaming Matrix Multiply Libraries on Intel platforms [22]. The following are the best published results from library implementations for conventional platforms, which differ from the experimental numbers from compilation found in Figure 29. The performance across platforms, measured in terms of FLOPS Per Cycle (FPC), ranges from 1.87 FPC on the Pentium 4 to 3.58 FPC on the Core 2 using SSE. The TRIPS application is able to achieve 5.20 FPC without the benefit of SSE, which is 40% greater than the best optimized code on the Core 2.

9.4.2 SPEC CPU2000

The right side of Figure 29 compares performance on SPEC2000 using reference data sets. TRIPS performance is much lower on the SPEC benchmarks than on the simple benchmarks. While floating-point performance is nearly on par with Core 2-gcc (Core 2-icc achieves a 1.9x speedup over TRIPS), integer performance is less than half that of the Core 2.

	Per 1000 useful instructions						
	Core 2 cond. br. misses	TRIPS cond. br. misses	TRIPS call/ret misses	Core 2 I-cache misses	TRIPS I-cache misses	TRIPS load flushes	Average useful insts in flight
bzip2	1.3	1.6	0.0	0.0	0.0	0.09	342.5
crafty	4.5	3.0	0.5	1.7	17.2	0.35	151.8
gcc	7.4	7.0	1.8	3.1	18.5	0.52	73.0
gzip	4.8	4.3	0.0	0.0	0.0	0.04	206.1
mcf	14.0	6.3	0.0	0.0	0.0	0.13	373.6
parser	2.0	3.2	0.1	0.0	0.6	0.04	—
perlbmk	2.5	0.4	8.3	0.0	13.0	0.19	106.9
twolf	8.5	4.8	0.1	0.0	8.2	0.36	275.2
vpr	0.5	1.4	0.5	0.0	3.2	0.40	221.8
ammp	0.2	1.5	0.1	0.0	1.0	0.05	—
applu	0.0	0.7	0.0	0.0	0.0	0.01	496.6
apsi	0.0	2.4	0.0	0.0	0.0	0.11	249.7
art	0.4	0.0	0.0	0.0	0.0	0.01	692.2
equake	0.2	0.6	0.0	0.0	0.9	0.08	337.9
mesa	1.4	1.6	0.0	0.0	3.5	0.04	199.4
mgrid	0.0	0.1	0.0	0.0	0.0	0.00	519.8
swim	0.0	1.0	0.0	0.0	0.0	0.00	416.1
wupwise	0.0	0.7	0.5	0.0	0.8	0.04	496.9

Table 10: Performance counter statistics for SPEC.

Table 11: Comparison of TRIPS Versus Commercial Processors from Figure 12 of [22].

Processor	**Kernel FPC**	**DGEMM FLOPS/cycle**	**# of Registers**	**L2 cache (KB)**
Opteron-EM64T	1.88	1.79	16x2	1024
P4-Prescott-EM64T	1.92	1.87	16x2	2048
Core2 Duo	3.68	3.58	16x2	4096
POWER5	3.84	3.78	32	1920
Itanium2	3.96	3.92	128	256
TRIPS	5.80	5.10	128	1024

Table 10 shows several events that have a significant effect on performance: conditional branch mispredictions, call-return mispredictions, I-cache misses, and load flushes for TRIPS, normalized to events per 1000 useful TRIPS instructions. Also shown are the branch mispredictions and I-cache misses for the Core 2, normalized to the same 1000 TRIPS instruction-baseline to ease cross-ISA comparison. The rightmost column shows the average useful TRIPS instructions in the window, from Figure 24.

Several of the SPECINT benchmarks have frequent I-cache misses, such as *crafty*, *gcc*, *perlbmk*, and *twolf*. These benchmarks are known to stress the instruction cache, and the block-based ISA exacerbates the miss rate because of TRIPS code expansion and the compiler's inability to fill the fixed-size 128-instruction blocks. *Perlbmk* also has an unusually high number of call/return mispredictions, due to an insufficiently tuned call and branch target buffer in TRIPS. All of these factors reduce the utilization of the instruction window; for example, *gcc* has an average of only 73 useful instructions in flight, out of a possible 121 based on the average block size. While the TRIPS call/return flushes and I-cache misses cause serious performance losses, branch mispredictions are competitive with the Core 2 and load dependence violations are infrequent. Benchmarks that have the most useful instructions in the window compare best to Core 2, such as *art* and *mgrid*. These benchmarks are known to contain parallelism, and show good performance with little compiler or microarchitectural tuning.

9.5 Application Study - DGEMM

This section elaborates a bit on the matrix multiply comparison shown in Figure 29. The matrix benchmark is an implementation of a Double-precision General Matrix Multiply (DGEMM). Our study in [13] describes optimization of

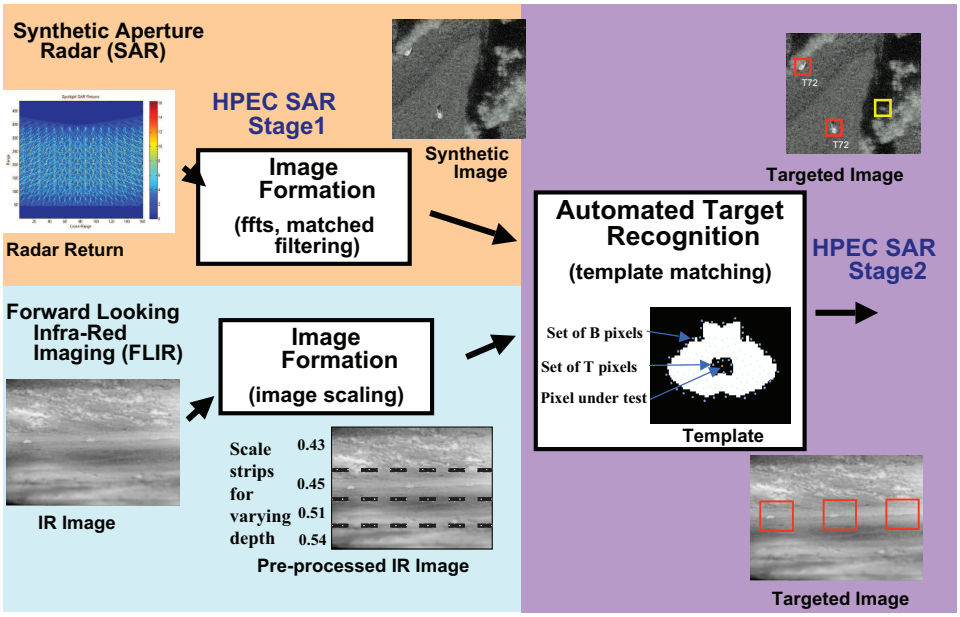


Figure 30: SAR End-to-End Application Description

DGEMM on the TRIPS platform. Algorithmically, we found that Goto's streaming approach continues to work well for moving data through the cache hierarchy [22], but algorithms for distributed uniprocessors such as TRIPS require some new approaches. We found that incorporating an extra level of matrix decomposition was necessary to make full algorithmic use of large register files and systolic-like communication between cache banks, register banks, and the ALU array. Further, we found that scheduling computation without consideration for operand network contention produced poor results. Instead, we mapped multiple vector-matrix multiplies across the spatial substrate to maximize resources and minimize network contention in the innermost loop. The resulting systolic-like algorithm nearly eliminates communication contention.

We also found that the costs of network communication versus local computation in on-chip networks makes performance sensitive to small changes in network and execution unit contention. Nonetheless, achieving high performance was relatively straightforward once we determined the relevant resource and algorithmic constraints. Our experience indicates that well-engineered algorithms will continue be relevant to emerging architectures that we anticipate seeing in the future. Our overall performance results are shown in Table 11. In addition to the sustained 5.8 FLOPS/cycle, the processor sustains a total of more than 11 instructions per cycle, when counting address computations and control instructions. This result shows the viability of data-driven architectures to exploit substantial instruction-level parallelism.

9.6 Application Study - SAR

The TRIPS project included an effort to evaluate TRIPS on library functions and applications of direct relevance to the DoD. The goal of this effort was to evaluate TRIPS not only in terms of performance, but in terms of maturity of the architecture and tools. We began with a survey of application requirements and concluded with execution and measurement of C functions and programs on the TRIPS prototype hardware. We conducted both low-level performance tuning and high-level code development using the TRIPS tools.

Our initial survey of embedded applications included a study of end-to-end applications and critical library functions. Our study of end-to-end embedded applications focused on synthetic aperture radar (SAR) processing and forward-looking infra-red imaging (FLIR) based on importance to DoD, computational requirements, and availability of benchmark code and data sets. Figure 30 shows an overview of these two processing chains. This survey led us to select the SAR processing benchmark developed by MIT Lincoln Laboratory as Scalable Synthetic Compact Application (SSCA) #3

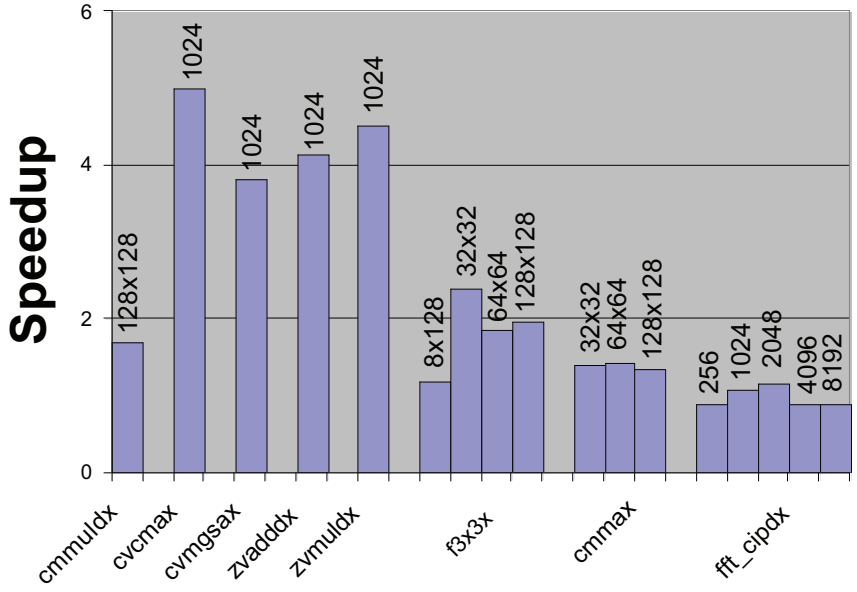


Figure 31: Library Performance Relative to PowerPC

under the DARPA High Productivity Computing Systems (HPCS) program [30]. We selected this application because it represented much of the functionality of both SAR and FLIR processing and because it is openly available within the DARPA community and was supported with C code and multiple data sets.

We selected key library functions based on a survey of the processing of a future fighter platform that would require communication, electronic warfare, phased array radar, and an electro-optical targeting and situational awareness sub-systems. Each of these sub-systems has different data types and rates and processing requirements, and performs different functions, and we used the types of processing functions typically used in these sub-systems to select functions from Mercury's Scientific Algorithm Library (SAL). The list of functions that we chose to investigate on TRIPS were: complex matrix multiplication (cmmuldx), vector conjugate multiply and add (cvcmax), vector magnitudes square and add (cvmgsax), complex vector add (zvadddx), vector multiply (zvmuldx), 3x3 convolution (f3x3x), complex matrix multiply and add (cmmax), and FFT (fft_cipdx).

Figure 31 shows the speedup results we obtained running on a single TRIPS core compared to a PowerPC/Altivec 7410 by comparing cycle counts on each processor. The PowerPC results are from the highly optimized vendor library implementation, which leverages the Altivec acceleration instructions. The results are normalized to factor out clock speed since clock speed is generally a function of implementation technology and design effort and methodology. The graph shows that TRIPS generally outperforms the PowerPC, by significant margins in some cases. TRIPS performs especially well on vector operations where the compiler can extract parallelism and the memory system provides relatively high bandwidth.

The methodology used to optimize these library functions was developed through experimentation and experience with the TRIPS tools. We started with a methodology based on attempting to manually map instructions to execution units in order to minimize communication and to avoid architectural bottlenecks. What we found is that the dynamic inter- and intra-block behavior combined with the complexity of scheduling approximately 100 instructions per basic block made it very hard to predict the outcome of manually scheduled instructions. As we learned this lesson and the TRIPS tools improved, we came to rely more on the TRIPS compiler and scheduler. We made extensive use of the critical path tool and analysis of TIL code to identify bottlenecks, but rather than manual optimization, we performed source code transformations to improve performance. Source code transformations included techniques that are often used with traditional computer architectures, such as: reducing cache misses by re-ordering computations and memory accesses

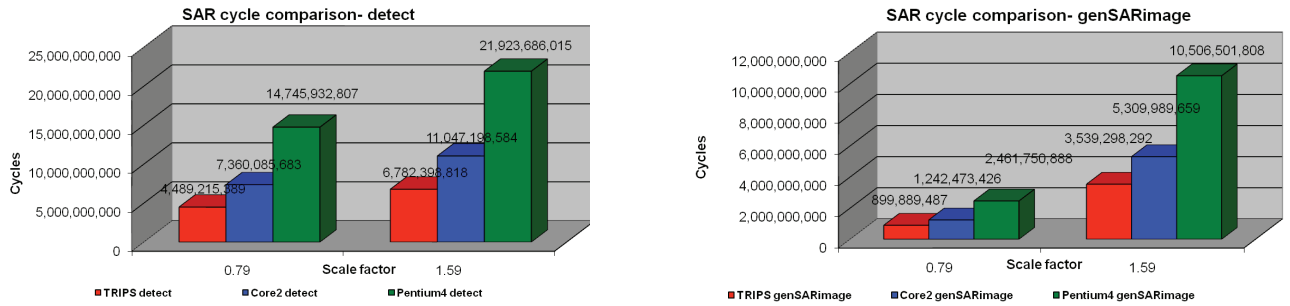


Figure 32: SAR Performance on TRIPS and Intel Architectures

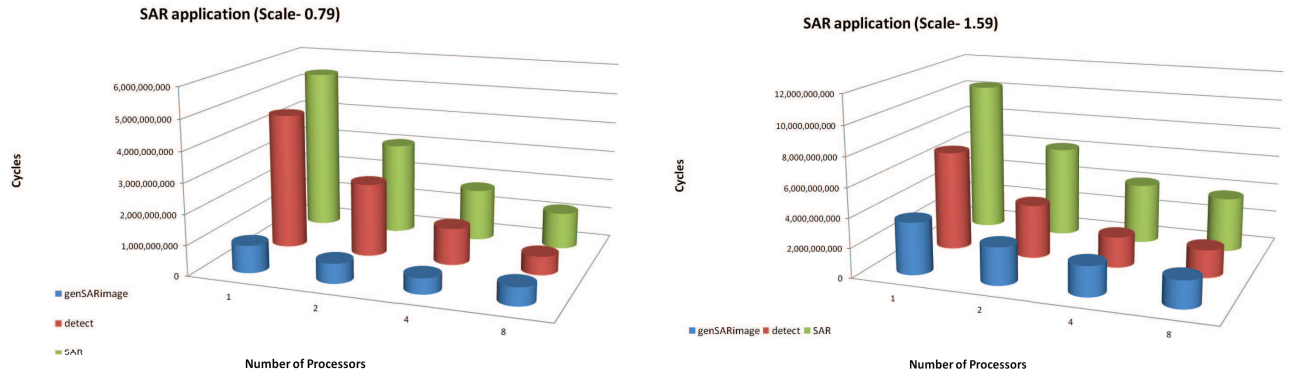


Figure 33: SAR MPI Performance on TRIPS

(blocking), reducing the number of memory accesses and unnecessary memory stores by re-ordering loops, distributing memory accesses between tiles (by padding arrays when necessary), exposing parallelism, and reducing overhead and unnecessary instructions (for example by boosting statements out of loops). These standard techniques allowed us to get excellent efficiency from the architecture that was competitive with or better than we did with manual optimizations, while improving the maintainability of the library functions.

Our performance analysis of the SAR benchmark included a performance comparison to Intel Core2 and Pentium4 processors and an implementation and performance analysis of an MPI version running on TRIPS. Performance was measured for the benchmark on two stages of the computation: genSARimage and detect. The benchmarks was executing using two data sizes, described as scale factors, of 0.79 and 1.59. Figure 32 shows the performance comparison of TRIPS with the Intel architectures, again using normalized cycle counts as the metric. TRIPS outperformed both architectures on both stages and data sizes. Previous results had indicated that TRIPS performed worse on genSARimage, but improvements in the tools over the last two years of the project improved performance considerably. This benchmark consists of thousands of lines of C code with several hundred lines in each of the timed stages, and we were able to compile and execute the application without modifying the source code (although we did modify the code to perform some performance optimizations). An application of this size demonstrates the robustness and maturity of the TRIPS tools.

We also developed a parallel MPI implementation of the benchmark, with both genSARimage and detect parallelized. Results of this implementation are shown in Figure 33, where the results are given in cycle counts. Results are shown for both stages of the application, as well as for the end-to-end application for both scale factors. These results show that we were able to get speedup up to 8 processors for the end-to-end application, and for all stages and data sizes, with the exception of the 8-processor implementation of detect for scale factor. In most cases, speedup is linear, or near linear. These results demonstrate the capability of the TRIPS MPI implementation and the functionality of the chip-to-chip network in a multi-processor system.

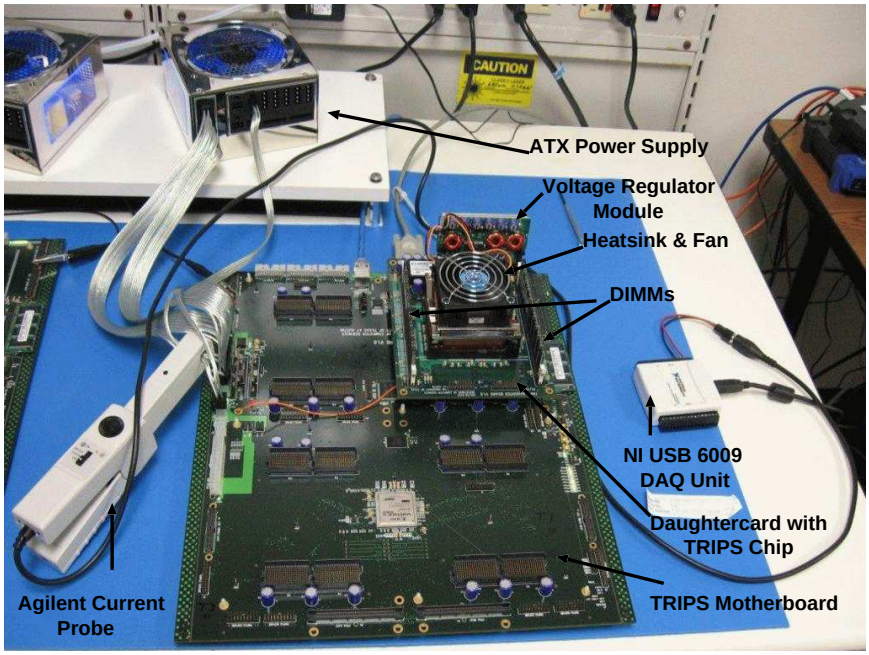


Figure 34: TRIPS Hardware Power Measurement Infrastructure

9.7 TRIPS Power Evaluation

To examine the power efficiency of TRIPS and TRIPS-like architectures, we first measured the power consumption of the TRIPS prototype. We then constructed an architectural-level power model, validated against our findings from the TRIPS power measurements and the TRIPS register transfer level (RTL) implementation in Verilog. Because the power consumed by the different elements of the TRIPS chip are not observable through experimentation, the architectural power model gives insight into the inherent tradeoffs associated with TRIPS and EDGE designs.

9.7.1 Hardware Power Measurement

Figure 34 shows a photograph of our power measurement infrastructure and the TRIPS prototype system. Each TRIPS motherboard can support up to four TRIPS chips. Each chip is mounted on the motherboard via a daughtercard. The daughtercard contains one Voltage Regulator Module (VRM) that steps down the 12 V ATX power supply to 1.5 V for the TRIPS chip, a heat-sink and fan assembly, and two 1-GB DDR SDRAM DIMMs. The DIMMs receive a 2.5 V power supply from the regulator. We use the following system parameters for our validation experiments: 1.5V chip power supply, 366 MHz chip clock frequency and 133/266 MHz for the DIMMs. We use an Agilent 1146A clamp-on current probe for measuring the power consumption of the TRIPS daughtercard. The voltage output of the probe is sampled by a National Instruments (NI) USB 6009 Data Acquisition System at the rate of 10 KHz and is logged to a Linux Workstation using the NI Data Logger program.

Using a set of carefully-designed experiments, we isolate the power consumed by the TRIPS motherboard and the DRAM DIMMs from the measured hardware power. Table 12 summarizes the measurable or derivable contributors to TRIPS system power. By removing the TRIPS daughtercards from the motherboard, we measure the motherboard power at 2.5 Watts. We also measure the daughtercard fan to consume 0.8W. To measure the power consumed by the DDR DIMMs, we take two power measurements: one when the DIMMs are unplugged from the daughtercard and a second when the DIMMs are plugged in and a sequence of random memory reads and writes are being performed. The difference between these two readings, about 3.6 Watts, is attributed to the DIMMs. After this isolation, we also account for 90% rated conversion efficiency of the VRM when converting from 12V to 1.5V needed for the TRIPS chip. The resulting power of 27.9 Watts is consumed by the TRIPS chip. To isolate the clock tree portion of the total power, we measure the dissipated

Table 12: Power Breakdown of a TRIPS System with a Single TRIPS Daughtercard at 1.5 Volts, 366 MHz, 130nm

Category	Power (W)
TRIPS Motherboard	2.5
Daughtercard Fan	0.8
Daughtercard DIMMs	3.6
Voltage Regulator Module	3.1
TRIPS Chip Clock Tree	18.3
Rest of TRIPS Chip	9.6
Total	37.9

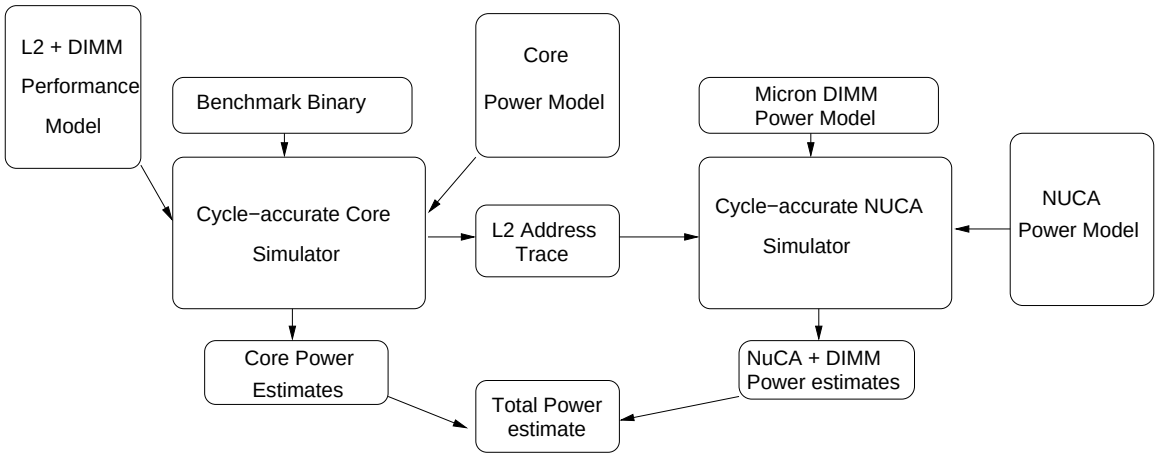


Figure 35: TRIPS Architectural Power Modeling

power of the chip when idling at 100, 200, and 366 MHz. We verify the first order linear dependence between clock frequency and power at these three points, and derive the clock tree power to be 18.3 Watts at 366 MHz. The remaining 9.6 Watts stems from the computation, memory, and communication on the chip.

9.7.2 Architectural Power Models and Validation

Figure 35 describes our baseline architectural power modeling methodology for TRIPS, which matches the state-of-the-art methodology in academia for building architectural power models. First, we run the benchmark binary on a cycle-accurate simulator that models the TRIPS processor core alone (excluding the L2). This simulation produces access counts of various microarchitectural structures in the core and a trace of all generated L2 addresses. Second, we run this L2 address trace through a cycle-accurate NUCA L2 simulator to obtain access counts of the structures in the L2 subsystem.

The TRIPS base architectural power is derived via commonly used power modeling methodologies. We build CACTI [66] models for all major structures such as caches, SRAM arrays, register arrays, branch predictor tables, load/store queue CAMs, and on-chip network router FIFOs to obtain a per-access energy for each structure. This per-access energy combined with the access counts from the architectural simulator provides the overall energy dissipated in these structures. The power models for integer and floating point ALUs and clock tree are derived from Wattch [6] using linear technology scaling from the built-in 350nm technology of Wattch. We model global clock drivers, global clock tree interconnect, pre-charge transistors and pipeline latches as part of the clock tree. We estimate the number of latches in each tile based on a detailed microarchitecture specification. The per-latch capacitance estimates are derived from Wattch as well.

To estimate control logic and interconnect power, we use rules-of-thumb to estimate the control logic gate counts and the average gate capacitance. From our experience with the TRIPS ASIC design, we assume the tile gate count is about four times the tile latch count. Given the gate counts, we use a proprietary rule-of-thumb in the IBM ASIC documentation to estimate the total gate capacitance. Using these estimates and models based on Rent's rule [65], we estimate the control

Table 13: TRIPS and Alpha 21264 Simulation Parameters

Parameter	Configuration
Microarchitecture	Alpha 21264
L2	4MB Cache (same as TRIPS)
Simulation	Sim-Alpha: A validated performance simulator combined with Wattch-like power models
Hand-optimized Benchmarks	3 kernels (conv, ct, genalg), 6 EEMBC benchmarks (a2time, autocore, basefp, dither, rspeed, tblook)
Compiled Benchmarks	5 kernels, 22 EEMBC benchmarks, 14 SPEC CPU benchmarks currently supported (8 Integer, 7 FP), simulated with single simpoints of 100 million cycles [59]

logic and interconnect access energies of the various tiles respectively. These energies combined with various event counts of the tiles provide the total energy dissipated for control logic and interconnect. We build leakage power models for all array structures based on HotLeakage [73]. Leakage power estimates for non-array structures are based on gate-count estimates and average transistor density estimates. We use an analytical power model from Micron [39] to estimate the DIMM power consumption for the architectural power models.

Our initial validation of the TRIPS architecture-level power model indicated that it underestimated the hardware's power consumption by 65%. We then used our Verilog RTL implementation to refine features of the architecture power models including clock tree overhead, gate count, and latch overhead. Ultimately, we improve the architecture power model's accuracy to within 15% of measured hardware power. Differences in the power models for control logic, interconnects, leakage, and the DIMMs cause the remaining discrepancy between modeled and measured power. More importantly, the relative accuracy of the architecture power model is within 3% of the hardware power consumption, where relative accuracy is a measure of the correlation in changes to power consumption in response to different programs. Specifically, the difference in the measured power between two different programs is within 3% across the hardware and the architecture-level power model. This observation bodes well for architecture studies that seek to compare relative power consumption across different applications and architecture configurations as long as the modeling, abstraction, and technology modeling errors in the power models are shared in common mode across the configurations.

9.7.3 Power Analysis and Comparison

To compare the power consumption of TRIPS to a commercial system, we used detailed simulators that capture the power consumed by the components within the processors. For TRIPS, we use our architecture power simulator, augmented with clock-gating to match that of a modern processor. Our clock gating models keep track of "active" cycles for various microarchitectural units. During active cycles, all units are attributed their full, ungated clock power. During every cycle that a unit is inactive, our models attribute a fixed percentage (about 30%) of the ungated clock power to that unit. No OPN routers in the TRIPS processor are ever clock gated to ensure correct operand communication. For the commercial system, we use the Alpha 21264 microprocessor [31].

Comparing the power of two different processors is challenging because one must factor out system-level components such as the memory and circuit boards. Such a comparison is even more challenging if insight into the power consumed by different microarchitecture elements of each chip is desired. We chose the Alpha system because we had already developed a cycle-accurate, performance-validated simulator for it called Sim-Alpha [12]. We then use Wattch-like power models built at 350nm and validated against published Alpha power data [23, 44] for power comparisons after suitable technology scaling to 130nm. Such detailed simulators for more recent chips, such as the Core family of processors, are not available. Nonetheless, the Alpha serves as a suitable reference system that illustrates the power advantages and disadvantages of the TRIPS system.

We mitigate the uncertainty in technology scaling by ensuring similar power modeling methodologies for all array structures and the ALUs for TRIPS as well as Alpha. Our simulation parameters for comparison with Alpha are listed in Table 13. To ensure a fair comparison we simulate a 4MB cache for both Alpha and TRIPS, although TRIPS employs a NUCA cache design. The table also lists the various benchmarks we use in this study. We use the SimPoint [59] methodology to choose a single SimPoint of about 100 million instructions from each of the SPEC benchmarks and simulate this region on the TRIPS and Sim-Alpha simulators.

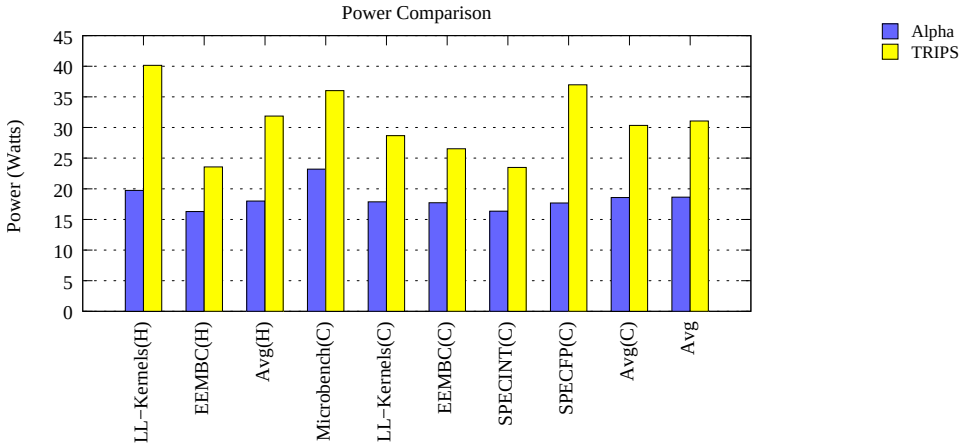


Figure 36: TRIPS Versus Alpha 21264: Total Power Comparison at 1.5V, 1 GHz, 130nm

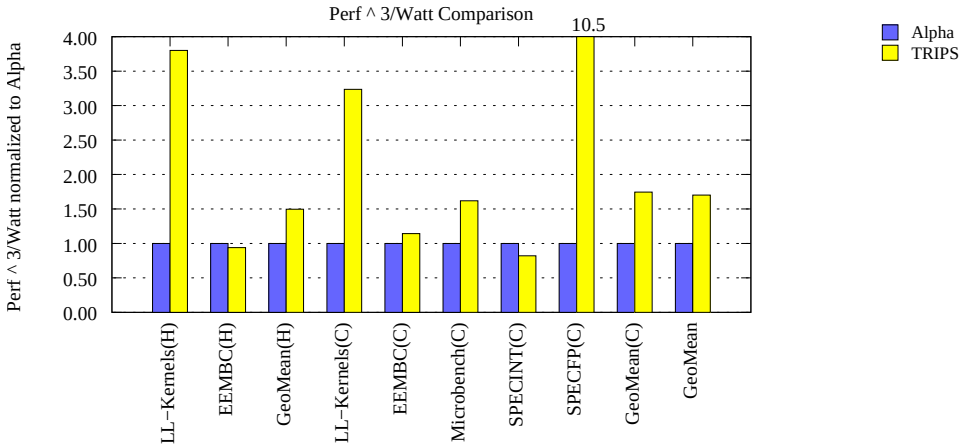


Figure 37: TRIPS Versus Alpha 21264: Performance^3/Watt Comparison

Figure 36 compares total power consumption of TRIPS and Alpha 21264. These results assume the following configuration: voltage supply of 1.5 Volts, 1GHz clock frequency and 130nm technology. The y-axis plots power dissipation in Watts and the x-axis shows various types of benchmarks. The graph shows benchmarks that are compiled and then hand-optimized on TRIPS (denoted by the letter 'H') separately from the ones that are purely compiled by the TRIPS compiler (denoted by 'C'). However, for Alpha, all these benchmarks are compiled with Alpha Gem compiler, whose code quality exceeds that of the current TRIPS compiler. We separate these benchmarks because hand-optimized benchmarks typically exhibit better performance on TRIPS than compiled ones. On an average we observe that TRIPS consumes 67% more power than Alpha 21264. Due to higher performance in hand-optimized benchmarks, hence higher activity in the microarchitectural units, power consumption in hand-optimized benchmarks is higher than compiled benchmarks.

Raw power is not a good metric of energy-efficiency, especially in high-end microprocessors like the Alpha and the TRIPS [5]. Since power depends on the processor voltage and frequency, we can reduce power dissipation by suitably adjusting the voltage and frequency. However, reducing voltage and frequency also affects the processor performance, which is not captured by the raw power metric. To simultaneously compare both power and performance in high-end processors, a metric called energy-delay-squared product or its inverse performance^3/watt should be used [5]. Figure 37 compares performance^3/watt of TRIPS and Alpha. The y-axis of the graph plots performance^3/watt normalized to that Alpha and the x-axis shows various types of benchmarks. The higher the values of performance^3/watt for a system the higher the energy efficiency. We can observe that the TRIPS achieves 1.7 times the energy-efficiency of Alpha in terms of performance^3/watt. On average TRIPS achieves better performance (measured in IPC) in all benchmark categories compared to Alpha, which results in better energy-efficiency of TRIPS despite consuming more power than Alpha.

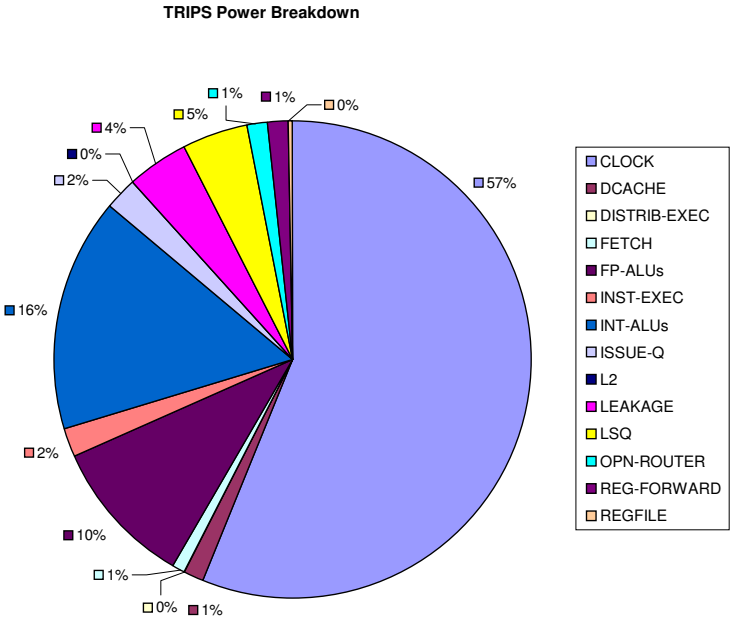


Figure 38: TRIPS Power Breakdown

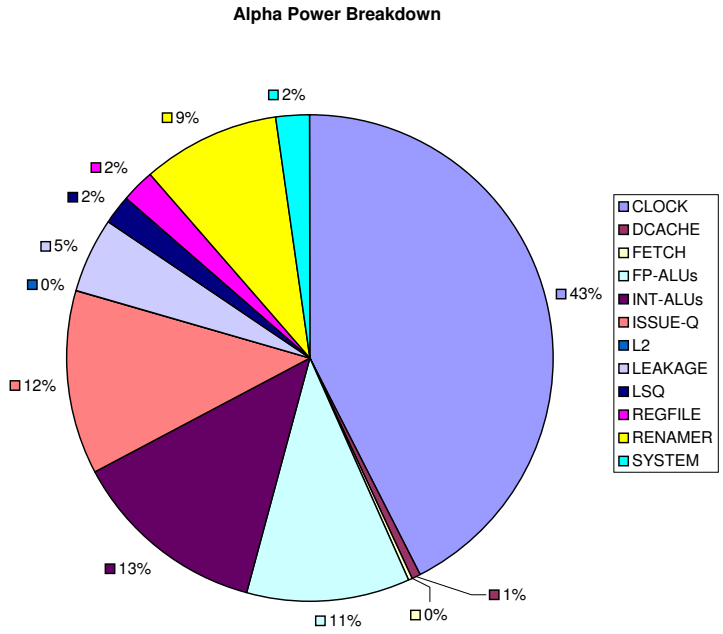


Figure 39: Alpha Power Breakdown

Figures 38 and 39 show the power breakdown of TRIPS and Alpha respectively. We use these detailed breakdowns to assess the power advantages and disadvantages of the TRIPS ISA and the TRIPS microarchitecture, recognizing that the comparison is not ideal as TRIPS is implemented using standard cells, while Alpha is implemented using custom circuits. The partitioned TRIPS microarchitecture implements simple register files with fewer ports compared to large, multi-ported register files in the Alpha. Thus, the register files in the Alpha consume almost seven times the power of the TRIPS register files (400 milliWatts compared to 60 milliWatts). Another major power advantage of the TRIPS ISA and microarchitecture is the explicit encoding of dependences in the ISA. This feature allows instruction issue windows to be constructed out of simple RAMs, and significantly simplifies the microarchitecture. In contrast, the Alpha microarchitecture performs associative searches in a CAM-based (Content Addressable Memories) issue window to identify dependent instructions. As a result, Alpha's issue window consumes about 3x the power of the TRIPS issue window.

The TRIPS ISA and microarchitecture have power disadvantages too. The TRIPS clock tree consumes about twice the power of the Alpha clock tree. The larger area of TRIPS (about 3x that of the Alpha) and the overheads of ASIC design methodology for TRIPS result in higher power dissipated in the clock tree. TRIPS instructions that are predicated out (about 25% of total), expansion of non-full TRIPS blocks into NOPs in the L1 I-cache, the operand network router, and maximally-sized load store queues in the data tiles are sources of power disadvantage for the TRIPS ISA and microarchitecture.

Despite these disadvantages, the 16-wide TRIPS microarchitecture extracts better performance than the Alpha to achieve better power efficiency. Furthermore, the runway for power and performance optimizations for TRIPS-like architectures is substantial. For example, while the large monolithic load/store queues in TRIPS consume substantial power, we have designed new distributed load/store protocols that reduce the area and power overheads of memory disambiguation logic [57]. Further, dynamically adapting the number of resources to the concurrency demands of a program can yield substantial power benefits. This form of hardware polymorphism is discussed in Section 10 in the context of configurable lightweight processor design.

10 Scalable and Configurable Systems

The TRIPS prototype was a first generation of EDGE-based microarchitectures and instruction sets. The TRIPS design showed how a large processor, much more aggressive than any built to date by industry, could be constructed out of distributed logic tiles, and connected by lightweight micronetworks. Part of our goal was to take the TRIPS design, learn from the prototyping experience, and propose a set of new technologies for advanced semiconductor processes that would scale in terms of power and performance. While the TRIPS design was novel and showed excellent performance on some highly concurrent workloads such as matrix multiply, the lessons we learned from the prototype drove the development of a second-generation EDGE architecture that we measured in simulation. In this section of the report, we describe this second-generation microarchitecture, called TFlex, and describe its additional polymorphism-supporting capabilities. We then describe the set of third-generation extensions and improvements to the architecture that we expect to provide further optimizations to performance and power efficiency.

10.1 Composable Lightweight Processors

The main additional polymorphous capability provided by the TFlex microarchitecture over the TRIPS microarchitecture (even though they use similar instruction sets) is the ability to be composable. The TFlex design is thus the first in a class of multicore designs called Composable Lightweight Processors, or CLPs.

A CLP consists of multiple simple, narrow-issue processor cores that can be aggregated dynamically to form more powerful single-threaded processors. Thus, the number and size of the processors can be adjusted on the fly to provide the target that best suits the software needs at any given time. The same software thread can run transparently—without modifications to the binary—on one core, two cores, up to as many as 32 cores in the design that we simulate. Low-level run-time software can decide how to best balance thread throughput (thread-level parallelism or TLP), single-thread performance (ILP), and energy efficiency. Run-time software may also grow or shrink processors to match the available ILP in a thread to improve performance and power efficiency. The polymorphous capability thus provided by CLPs is dynamic heterogeneity: permitting the processors to select the number and granularity of processors at any given time, thus meeting the requirements of whatever software happens to be running at the time.

Figure 40 shows a high-level floorplan with three possible configurations of a CLP. The small squares on the left of each floorplan represent a single processing core while the squares on the right half show a banked L2 cache. If a large number of threads are available, the system could run 32 threads, one on each core (Figure 40(a)). If high single-thread performance is required and the thread has sufficient ILP, the CLP could be configured to use an optimal number of cores that maximizes performance (up to 32, as shown in Figure 40(c)). To optimize for energy efficiency, for example in a data center or in battery-operated mode, the system could configure the CLP to run each thread at its best energy-efficient point. Figure 40(b) shows an energy-optimized CLP configuration running eight threads across a range of processor granularities.

A fully composable processor shares no structures physically among the multiple processors. Instead, a CLP relies on distributed microarchitectural protocols to provide the necessary fetch, execution, memory access/disambiguation, and commit capabilities. CLPs are a good match for EDGE architectures, as full composability is difficult in conventional

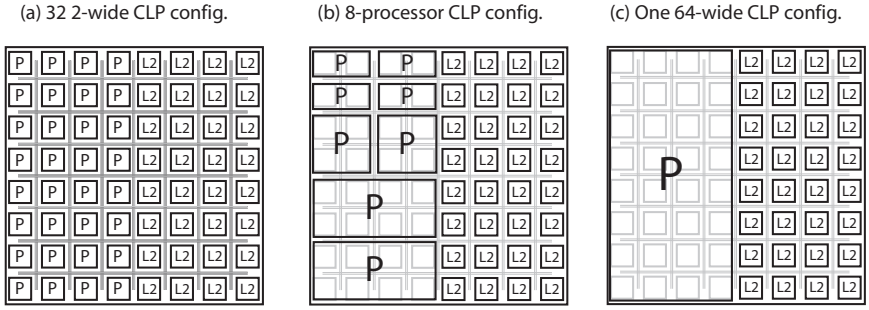


Figure 40: Three Dynamically Assigned CLP Configurations

ISAs. With individual instructions as atomic units, control decisions must be made too frequently to coordinate across a distributed processor.

Unlike in the TRIPS microarchitecture, a TFlex microarchitecture has no centralized structures (such as the G-tile), so every single microarchitectural function must be distributed. Below we describe the innovations that we incorporated into the microarchitecture to achieve this capability.

A composable processor must allow its microarchitectural structures to linearly increase (or decrease) in capacity as participating cores are added (or removed). For example, doubling the number of cores should double the number of useful load/store queue entries, the usable state in the branch predictors, and the cache capacities. The CLP microarchitecture partitions structures by address whenever possible, and avoids physically centralized microarchitectural structures completely.

This complete partitioning addresses some of the limitations of the original TRIPS microarchitecture. Specifically, the next-block predictor state and the number of data cache banks were limited by the centralization of the predictor and the load-store queue, respectively. Full composability necessitates distributing those structures as well, which provides higher overall performance than the TRIPS microarchitecture irrespective of the composable capabilities. However, those performance gains are a side benefit to the significantly increased flexibility that composition provides.

The TFlex microarchitecture uses three distinct hash functions for interleaving across three classes of structures:

- Block starting address: The next-block predictor resources (e.g., BTBs and local history tables) and the block tag structures are partitioned based on the starting virtual address of a particular block, which corresponds to the program counter in a conventional architecture. Predicting control flow and fetching instructions in TFlex occurs at the granularity of a block, rather than individual instructions.
- Instruction ID within a block: A block contains up to 128 instructions, which are numbered in order. Instructions are interleaved across the partitioned instruction windows and instruction caches based on the instruction ID, theoretically permitting up to 128 cores each holding one instruction from each block.
- Data address: The load/store queue and data caches are partitioned by data address from load/store instructions, and registers are interleaved based on the low-order bits of the register number.

In addition, register names are interleaved across the register files. However, because a single core must have 128 registers to support single-block execution, register file capacity goes unused when multiple cores are aggregated. Because interleaving is controlled by bit-level hash functions, the number of cores that can be aggregated to form a logical processor must be a power of two.

10.1.1 Overview of TFlex Operation

While a portion of each in-flight instruction block is assigned to each participating core, a block is assigned a single owner core, based on a hash of the block address, which is responsible for initiating fetching of the block and predicting the next block. Once the next-block address is predicted, the owner core sends that address to the core that owns the next predicted block. Each owner core is also responsible for launching pipeline flushes of mis-speculations caused by its block, for detecting that its block is complete, and for then committing it.

Figure 41 provides an overview of TFlex execution for the lifetime of one block. It shows two threads A and B running on eight cores each; for each thread, two blocks (A0/A1 and B0/B1) are in flight. In the block fetch stage, the block owner accesses the I-cache tag for the current block and broadcasts the fetch command to the I-cache banks in the participating cores (Figure 41(a)). In parallel, the owner core predicts the next block address and sends a control message to the next block owner to initiate fetch of the subsequent block (Figure 41(b)). Up to eight blocks may be in flight for eight participating cores. Upon receiving a fetch command from a block owner, each core fetches its portion of the block from its local I-cache, and dispatches fetched instructions into the issue window. Instructions are executed in dataflow order

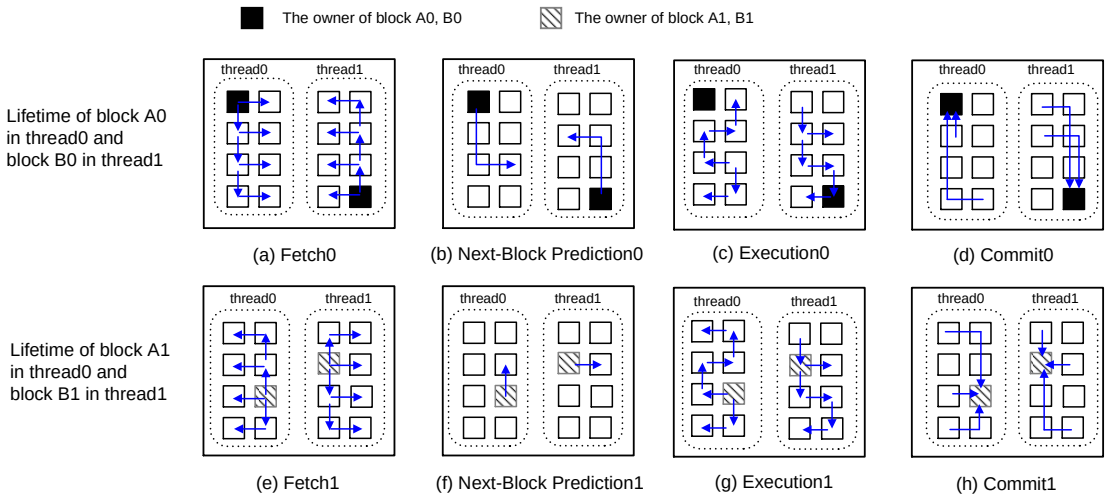


Figure 41: TFlex Execution Stages

when they are ready (Figure 41(c)). When a block completes, the owner detects completion, and when it is notified that it holds the oldest block, it launches the block commit protocol, shown in Figure 41(d). Figures 41(e-h) show the same four stages of execution for the next block controlled by a different owner; fetch, execution, and commit of the blocks are pipelined and overlapped. Finally, the diagrams show that two distinct programs can be run on non-overlapping subsets of the cores.

10.1.2 Composable Instruction Fetch

In the TFlex microarchitecture, instructions are distributed across the private I-caches of all participating cores, but fetches are initiated by each of the instruction block's owner core. The owner core manages the tags on a per-block basis, and functions as a central registry for all operations of the blocks it owns. The cache tags for a block are held exclusively in the block owner core. In a 32-core configuration, each core caches four instructions from a 128-instruction block.

With this fetch model, the fetch bandwidth and the I-cache capacity of the participating cores scale linearly as more cores are used. While this partitioned fetch model amplifies fetch bandwidth, conventional microarchitectures cannot use it because current designs require a centralized analysis point of the fetch stream (for register renaming and inum assignment) to preserve correct sequential semantics. This constraint poses the largest challenge to composable processors built with conventional ISAs. In contrast, EDGE ISAs overcome these deficiencies by explicitly and statically encoding the dependence order of the instructions within a block; the dynamic total order among all executing instructions is obtained by concatenating the block order and the statically encoded order within a block. Given large block sizes, distributed fetching is feasible because instruction dependence relations are known a priori.

10.1.3 Composable Control-flow Prediction

Control-flow predictors are some of the most challenging structures to partition for composability, since the predictor state has traditionally been physically centralized to facilitate few cycles between successive predictions. The TFlex composable predictor treats the distributed predictors in each composed core as a single logical predictor, exploiting the block-atomic nature of the TRIPS ISA to make this distributed approach tenable. Similar to the TRIPS prototype microarchitecture, the TFlex control flow predictor issues one next-block prediction for each 128-instruction hyperblock—a predicated single entry, multiple exit region—instead of one prediction per basic block.

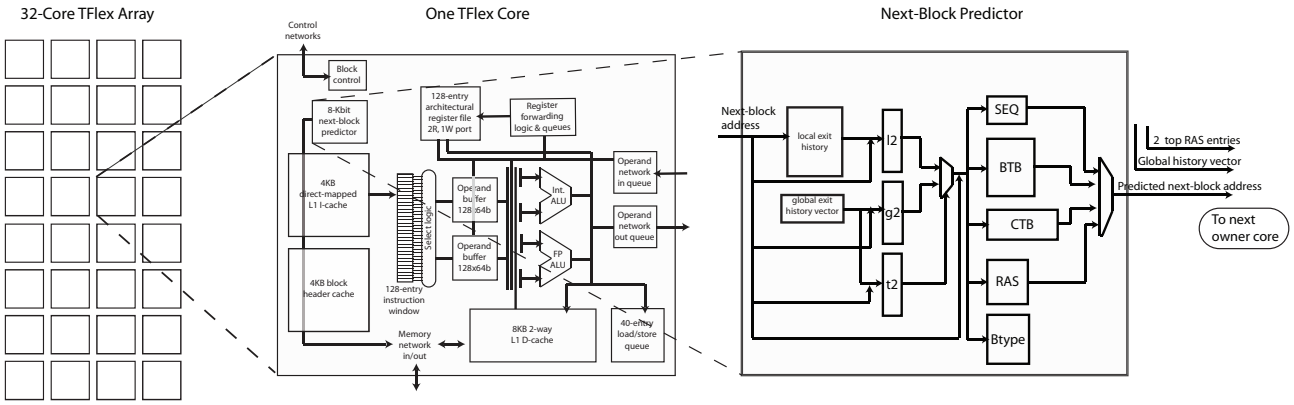


Figure 42: TFlex Core Microarchitecture

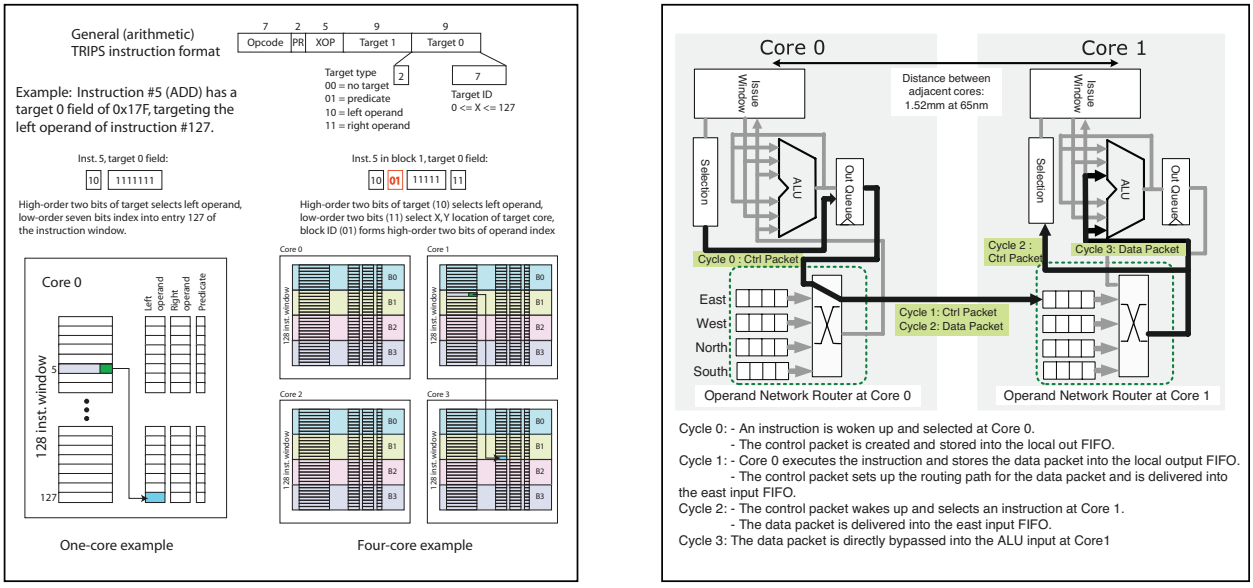
Figure 42 illustrates the distributed next block predictor consisting of eight main structures. Each core has a fully functional block predictor and the predictors are identical across all the cores. The block predictor can be divided into two main components: the exit predictor, predicting which branch will be taken out of the current block; and the target predictor, which predicts the target address of the next block. Each branch in a block contains three exit bits in its instruction, which are used to form histories instead of the traditional taken/not taken bits.

The exit predictor is an Alpha 21264-like tournament-style hybrid predictor and is composed of traditional two-level local, global, and choice predictors that use local and global block exit histories. The local as well as global histories are updated speculatively after a prediction and repaired from backup history buffers on a misprediction. The target address is predicted by first predicting the type of the exit branch—a call, a return, a next sequential block, or a regular branch—using the Btype predictor. The Btype predictor result selects one of those four possible next-block targets, which are provided by a next-block adder (SEQ), a Branch Target Buffer (BTB) to predict branch targets, a Call Target Buffer (CTB) to predict call targets, and a Return Address Stack (RAS) to predict return targets.

Each of the major predictor structures is affected by the composed nature of the microarchitecture. The local histories are trivially composable since, for a fixed composition, the same block address will always map to the same core. Local predictions do not lose any information even though they are physically distributed. Similarly, the Btype, BTB and CTB tables hold only the target addresses of the blocks owned by that core. With this organization, the capacity of the predictor increases as more cores are added, assuming that block addresses tend to be distributed among cores equally.

The global predictor is more complex because of the distributed history. When a predicted next-block address is sent from the previous blocks' owner to the owning core of the predicted block, the global exit history is forwarded along with the prediction. This forwarding enables each prediction to be made with the current global history, without additional latency beyond the already incurred point-to-point latency to transmit the predicted next-block address from core to core.

The component that is most difficult to distribute, the Return Address Stack (RAS), must be maintained as a single logical stack across all the cores since it represents the program call-stack. The TFlex microarchitecture uses the composed entries to permit a deeper RAS. Instead of using address interleaving, it sequentially partitions the RAS across all the cores (e.g., a 32-entry stack for 2 cores would have entries 0 to 15 in core 0 and 16 to 31 in core 1). The stacks from the participating cores form a logically centralized but physically distributed global stack. If the exit branch type is predicted as a call, the corresponding return address is pushed on to the RAS by sending a message to the core holding the current RAS top. If the branch is predicted as a return then the address on the stack is popped off by sending a pop-request to the core holding the RAS top. Recovery upon a misprediction is the responsibility of the mispredicting owner, which rolls back the mis-speculated state and sends the updated histories and RAS pointers to the next owner core, as well as the corrected top-of-stack RAS information to the core that will hold the new RAS top.



(a) (b)

Figure 43: (a) Block Mapping for One-core and Four-core Processors, (b) Inter-core Operand Communication

10.1.4 Composable Instruction Execution

As previously discussed, each instruction in an EDGE block is statically assigned an identifier between 0 and 127 by the TRIPS compiler. A block's instructions are interleaved across the cores of a composable processor using a mapping function. Changing the mapping function when cores are added (or removed), achieves composability of instruction execution. Figure 43(a) shows the mechanism that the TFlex design uses to support dynamic issue across a variable number of composed cores. Each instruction in an EDGE ISA block contains at least one nine-bit target field, which specifies the location of the dependent instruction that will consume the produced operand. Two of the nine bits specify which operand of the destination instruction is targeted (left, right, or predicate), and the other seven bits specify which of the 128 instructions is targeted. Figure 43(a) shows how the target bits are interpreted in single-core mode, with instruction five targeting the left operand of instruction 127. In single-core mode, all seven target identifier bits are used to index into a single 128-instruction buffer.

Figure 43(a) also shows how the microarchitecture interprets the target bits when running in a four-core configuration. The four cores can hold a total of four instruction blocks, but each block is striped across the four participating cores. Each core holds 32 instructions from each of the four in-flight blocks. In this configuration, the microarchitecture uses the two low-order bits from the target to determine which core is holding the target instruction, and the remaining five bits to select one of the 32 instructions on that core. The explicit dataflow target semantics of EDGE ISAs make this operation simple compared to what would be required in a RISC or CISC ISA where the dependences are unknown until an instruction is fetched and analyzed.

The latency incurred in routing dependent operands from core to core influences performance greatly. The TFlex cores are connected by a two-dimensional mesh network; bypassing an operand between adjacent cores incurs only a single-cycle bubble as a control message is sent one cycle in advance of the data message to wake up the target instruction. Figure 43(b) shows the datapath from the output of an ALU in one core to the input of an ALU in an adjacent core and illustrates cycle-by-cycle activities when the execution result at core 0 is bypassed into core 1. Area estimates for 65nm indicate a core-center to core-center distance of 1.5mm, corresponding to an optimally repeated wire delay of 170ps. With a fast router that matches the wire delay, the total path delay would be less than 350ps, enabling a one-cycle inter-core hop latency to be supported at over 2.5GHz.

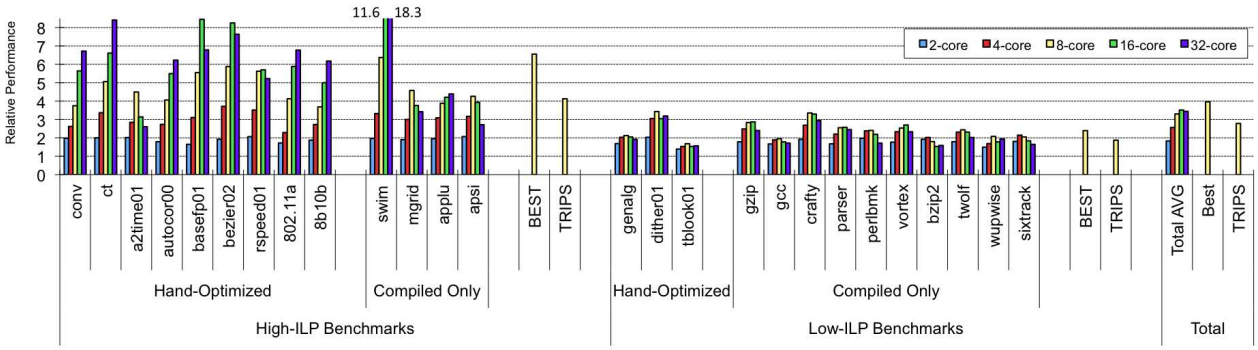


Figure 44: TFlex Performance for 2 to 32 cores, Normalized to a Single TFlex Core

10.1.5 Composable Memory Disambiguation

One of the primary challenges we solved with respect to CLPs was that of memory ordering: maintaining the correct execution in the face of potentially hundreds of loads and stores in flight simultaneously, many to the same addresses. We developed a technique called unordered load/store queues, which reduces both the load/store queue size in conventional microarchitectures, but also permits the load/store queues to be heavily banked, thus making them distributable and address-interleavable across participating composed processors [57].

10.1.6 Composable Dependence Prediction

The last distributed microarchitectural protocol invented by this work supports aggressive dependence prediction. When many loads and stores are in flight, some of them may resolve to the same address, requiring loads to wait until earlier stores to the same address have resolved. However, significant performance is lost if the loads wait very long. Dependence prediction speculates on which loads will be independent of all in-flight stores (and can thus safely issue immediately when ready), and which loads will be dependent, and thus must be held back from executing for some time. Store Sets, the state of the art in dependence predictors, cannot function well in a distributed/tiled environment. We proposed a new class of dependence predictors, close to the accuracy of Store Sets, despite their inability to monitor a centralized fetch stream. These counting dependence predictors increase performance significantly and are necessary for good performance on a CLP [52].

10.1.7 TFlex Performance

Figure 44 shows the performance of the TRIPS prototype architecture and that of TFlex configurations ranging from 2 to 32 cores, normalized to the performance of a single TFlex core. The 26 benchmarks on the x-axis are arranged into categories of low and high IPC. On average, the 16-core TFlex configuration performs best and shows a factor of 3.5 speedup over a single TFlex core. When the processor is configured to the best performing number of cores for each application (represented by the bar “BEST”), the performance of TFlex increases an additional 13% and the overall speedup over a single TFlex core reaches a factor of four. These results indicate that, using the proposed execution model, sequential applications can be effectively run across multiple cores to achieve substantial speedups.

On average, an eight-core TFlex processor, which has the same area and issue width as the TRIPS processor, outperforms TRIPS by 19%, reflecting the benefits of additional operand network bandwidth as well as twice the L1 cache bandwidth stemming from fine-grained distribution of the cache. Using the best per-application TFlex configuration outperforms TRIPS by 42%, demonstrating that adapting the processor granularity to the application granularity provides significant improvements in performance.

10.2 TRIPS Architecture and Microarchitecture Enhancements

The second-generation TFlex design showed improved polymorphous capabilities, better power efficiency, and better overall performance than the TRIPS prototype. However, there are a number of additional enhancements that we have developed and which have been partially evaluated. These enhancements will increase the performance and power efficiency of polymorphous, composable architectures.

10.2.1 Predicate Prediction

One of the discoveries we made when evaluating the TRIPS prototype system was that predication lowered performance far more than we thought likely. When a basic block is predicated by the TRIPS compiler, a branch is converted to a test instruction, which produces a predicate that will enable or disable some instructions' execution. If the original branch would have been successfully predicted, that condition does not affect performance at all. However, the run-time realization of the predicate often ends up on the critical path.

We have evaluated a predicate prediction scheme for the TFlex microarchitecture. Predicates that are at the head of predicate chains can be predicted immediately after fetch. The predicted predicate is sent to their consumers, and if one of those consumers is a test instruction, it will then also be predicted. Each prediction must communicate with the block owner core to ensure that all speculations have been verified by the time the block can first be committed. This predicate prediction scheme uses small tables (2KB), and achieves an average of 20% performance improvement, compared to 40% if perfect predicate prediction is assumed. We believe that predicate prediction will be a critical and necessary part of any future EDGE-based composable microarchitecture.

10.2.2 Alternate Dataflow Graph Mappings

The TRIPS prototype used a fixed mapping strategy of instruction blocks to execution tiles. The TFlex microarchitecture permitted the number of cores (and thus execution units) to vary dynamically, but not the mapping of instruction blocks to the available cores. We investigated alternative mapping strategies, to best balance concurrency and communication.

At one extreme, a flat mapping strategy spreads each block across all of the available cores. This strategy provides the greatest opportunity for parallelism within a block, yet incurs the highest intra-block communication overheads. The instruction identifiers assigned by the compiler are used by the block mapper for both locality and criticality information with this mapping strategy. At the other extreme, a deep mapping strategy assigns each entire block to a single core. With this strategy the parallelism within a block is limited to the core's issue width, but there is no communication overhead between instructions within a block, except when those instructions access memory. The mapper uses the instruction identifiers only for criticality. With a deep mapping strategy, the block mapper assigns all instructions within a block to a single core. This strategy eliminates cross-core communication between instructions, but provides only as much intra-block parallelism as the issue width of the cores. Although deep mapping eliminates communication between instructions, it may increase communication between blocks because cache banks and registers are distributed across the cores, while each block executes on only a single core.

The fixed strategies are limited because each block has the same opportunity for parallelism, yet blocks have different concurrency characteristics. For adaptive strategies, the compiler encodes an abstract concurrency value in the ISA for each block. The mapper assigns cores as necessary to exploit the available concurrency, yet avoids incurring unnecessary communication overheads. With a flat mapping strategy, the block mapper distributes instructions for each block across all participating cores using instruction identifiers set by the compiler. This approach exploits as much concurrency within a block as possible, but incurs intra-block communication costs between instructions.

With the deep and adaptive mappings, nearly 70% of the operand network traffic is eliminated, which is one of the major sources of energy overhead in the original TFlex design. In addition, performance is increased by an average of over 15% over the flat mapping, assuming dual-issue TFlex cores. The dynamic mapping of blocks to cores also simplifies the

instruction-fetch front end considerably [51].

10.2.3 Operand Network Optimizations

One of the largest sources of overhead in the TRIPS ISA is the presence of many `move` instructions for wide fanout instructions (i.e., instructions that have many targets within a block). We have developed a hybrid RAM/CAM instruction window, which incorporates instruction set changes to support the identification of instructions that must send a target to many instructions, and those that must receive it. By defining a number of broadcast identifiers for each block, the wide fanout instruction may send its result to a broadcast bus connected to CAMs for each instruction. Instructions waiting for a broadcast will be monitoring the bus for their broadcast ID. When that ID is sent, the waiting instructions remove the value from a broadcast bus. This structure is similar to the wakeup logic in a conventional superscalar processor, but significantly more efficient, as only the instructions waiting for a broadcast are consuming power on the narrow broadcast bus and CAMs. An additional necessary feature is the inclusion of multicast support through the operand network; if a wide fanout instruction needs to send operands to two adjacent cores whose routes share a link, it is far more efficient to send one message where the payload is split when the message reaches a divergence point. Using this technique, we have shown that average performance gains of 18% can be achieved, while eliminating approximately 65% of all `move` instructions.

10.3 Summary

We have developed and evaluated a succession of microarchitectural improvements to the initial TRIPS approach. The performance on hand-optimized assembly, as well as our estimates of power for TFlex, are far better than even the best commodity processors. Furthermore, when the above optimizations are all incorporated into the same experimental TFlex microarchitecture, we believe that with 8 or 16 composed cores, complex integer codes compiled using the TRIPS compiler will outperform the Core2 (cycle for cycle) using Intel's native compiler, with much better power efficiency. With a production-quality compiler, it is likely that performance would be significantly better. The scaled and configurable systems we have developed using the experience gained from the TRIPS prototype now show qualitatively better flexibility (adapting the hardware to the number of available threads), as well as improved performance and power efficiency that we believe will be significantly better when this polymorphous technology is adapted into production systems. It is possible that an industrial production of this polymorphous EDGE architecture will see major improvements over the state of the art, such as a five-fold improvement in energy-delay squared compared to the very best commercial parts.

11 Summary

The prototyping effort's goals were twofold: to determine the viability of EDGE technology and to learn the right way to build an EDGE-based machine. This design and evaluation effort taught the following specific lessons about how this class of architectures should be built.

EDGE ISA: Prototyping has demonstrated that EDGE ISAs can support power-efficient, large-window, out-of-order execution with less complexity than an equivalent superscalar processor. However, the TRIPS ISA had several significant weaknesses. Most serious was the limited fanout of the `move` instructions, which results in far too many overhead instructions for high-fanout operations. The ISA needs support for limited broadcasts of high-fanout operands. In addition, the binary overhead of the TRIPS ISA is too large. The 128-byte block header, with the read and write instructions, adds too much per-block overhead. Future EDGE ISAs should shrink the block header to no more than 32 bytes, and must support variable-sized blocks in the L1 I-cache to reduce the NOP bloat, despite the resultant increase in microarchitectural complexity.

Compilation: The TRIPS compiler and prototype has shown that correct EDGE code can be generated, even for complex integer applications. The hand optimizations that proved effective are largely mechanical, indicating that a production EDGE compiler could achieve much of that improvement. Because of the instruction-level block constraints, we determined that structural optimizations, such as loop unrolling and hyperblock formation, should occur in the back end after code generation. In general, the ISA model faces several difficult compilation challenges, the most significant of which is forming large blocks in control-intensive code. The major challenge is frequent function calls that cut blocks too early; inlining cannot solve this problem because it occurs well before block formation, typically in the front end. A second critical problem is allocating as many variables in registers as possible; the best hand-generated code replaced store-load pairs with a intra-block temporary communications, producing tighter code and higher performance. Effective interprocedural alias analysis is required to discover sufficient opportunities for this optimization.

Microarchitecture: The TRIPS prototype demonstrates that a microarchitecture with distributed protocols is feasible, and the fully functional first silicon indicates that tiled architectures benefit from increased design and validation productivity. A positive result was that, in general, the distributed block control protocols (fetch, dispatch, commit, flush) are not on the critical path. However, a number of artifacts in the microarchitecture resulted in significant performance losses. Most important was traffic on the operand network, which averaged just under one hop per operand. That amount of communication resulted in both significant OPN contention and communication cycles on the critical path. Spreading a block's instructions among all execution tiles caused too much intra-block communication. Follow-on microarchitectures must re-map instructions, in coordination with the compiler, so that most instruction-to-instruction communication occurs on the same tile. The second most important lesson was that performance losses due to the evaluation of predicate arcs was occasionally high, since arcs that could have been predicted as branches are deferred until execution. Future EDGE microarchitectures must support predicate prediction to evaluate the most predictable predicate arcs earlier in the pipeline. Third, the primary memory system must be distributed among all of the execution tiles; the cache and register bandwidth along one edge of the execution array was insufficient for many bandwidth-intensive codes. Finally, a minor design flaw in the prototype was making the call/return predictors too small, which should be enlarged in future microarchitectures. Improvements in branch and dependence predictors will also result in higher performance for all microarchitectures, including EDGE designs.

12 Recommendations

The TRIPS project has demonstrated through hardware and software prototypes that distributed and configurable chip architectures are implementable and viable. The TRIPS NUCA cache can be shared, partitioned, or even be converted in part to scratchpad storage. EDGE instruction set architectures and corresponding microarchitectures are in fact feasible to build; TRIPS demonstrates a tiled design that exploits out-of-order execution over a window of many hundreds of instructions. Despite the inter-tile routing latencies, the combination of the large window, dynamic issue, and highly concurrent memory system permits TRIPS to sustain up to ten instructions per cycle, and substantial speedup opportunities over commercial microprocessors. Repartitioning the responsibilities between the compiler and the hardware results in the elimination of power hungry structures in conventional architectures. The custom compiler for the TRIPS prototype demonstrates the capability to translate and optimize programs for data-driven computer systems.

Furthermore, the TRIPS implementation is merely the first EDGE implementation; our research shows that there is a substantial runway ripe for performance improvements in potential successive generations of TRIPS-like processors. While some of the improvements are microarchitectural or instruction set enhancements, other are more fundamental. The configurable lightweight processors architecture can employ an EDGE instruction set, but provides polymorphous capabilities to end applications. Reconfiguration can occur transparently to the application, based on changes to the applications characteristics or environment, or configuration can be controlled directly through application or operating system software. Essentially, the hardware can be configured (by itself or with operating system support) to meet the needs of whatever software workload mix is running on it, providing true polymorphous capabilities.

Looking forward, we see tremendous potential for TRIPS-like architectures. To bring them to fruition in the marketplace will require both research and engineering effort. While the TRIPS prototype compiler is effective, we are not satisfied with the quantity of parallelism it can extract or the quality of code it generates. Further investment in compiler technologies for distributed and configurable architectures will be critical to their success. While the architectures of the TRIPS project demonstrate the viability of polymorphous systems, we expect that additional potential lies with dynamic specialization of architectures through runtime configuration. Making this vision successful will require breakthroughs in microarchitecture, hardware virtualization, and software scheduling. Finally, modern architectures are saddled with outdated instruction set architectures which prevent many performance and efficiency oriented optimizations. Investments into technologies that can break the chains of legacy ISAs is necessary to enable architecture innovations that will help satisfy both commercial and DOD needs. While it is possible that TRIPS-like capabilities will be folded into conventional microarchitectures, similar to how Intel co-opted the design advantages of RISC ISAs with their P6 microarchitecture, we think it more likely that this technology can only achieve its potential with either direct native compilation to EDGE ISAs or dynamic translation/compilation. Given the twin parallelism and power crises facing industry, we expect commercial interest to grow with each passing month.

13 References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *International Symposium on Computer Architecture*, pages 227–237, June 1998.

[3] J. Backus, R. Beeber, S. Best, R. Goldberg, L. Haibt, H. Herrick, R. Nelson, D. Sayre, P. Sheridan, H. Stern, I. Ziller, R. Hughes, and R. Nutt. The Fortran Automatic Coding System. In *Proceedings of the Western Joint Computer Conference*, pages 187–198, February 1957.

[4] C. Bastoul. Generating Loops for Scanning Polyhedra. Technical Report 2002/23, PRiSM, Versailles University, 2002.

[5] D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P. Cook. Power-aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. *IEEE Micro*, 20(6):26–44, November/December 2000.

[6] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-level Power Analysis and Optimizations. In *International Symposium on Computer Architecture*, pages 83–94, May 2000.

[7] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[8] K. E. Coons, X. Chen, D. Burger, K. S. McKinley, and S. K. Kushwaha. A Spatial Path Scheduling Algorithm for EDGE Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, October 2006.

[9] K. E. Coons, B. Robatmili, M. E. Taylor, B. A. Maher, D. Burger, and K. S. McKinley. Feature Selection and Policy Optimization for Distributed Instruction Placement Using Reinforcement Learning. In *International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[10] DDD - Data Display Debugger. Free Software Foundation, 2005. `http://www.gnu.org/software/ddd`.

[11] W. Denk. U-Boot - Open Source Firmware for Embedded PowerPC, 2007. `http://www.denx.de`.

[12] R. Desikan, D. Burger, and S. W. Keckler. Measuring Experimental Error in Microprocessor Simulation. In *International Symposium on Computer Architecture*, pages 266–277, July 2001.

[13] J. Diamond, B. Robatmili, S. W. Keckler, K. Goto, D. Burger, and R. van de Geijn. High Performance Dense Linear Algebra on Spatially Partitioned Processors. In *Symposium on Principles and Practice of Parallel Programming*, pages 63–72, February 2008.

[14] D. W. Doerfler. An Analysis of the Pathscale Inc. InfiniBand Host Channel Adapter, InfiniPath. Technical Report SAND2005-5199, Sandia National Laboratories, August 2005.

[15] The Embedded Microprocessor Benchmark Consortium. `http://www.eembc.org`.

[16] A. Faraj and X. Yuan. Communication Characteristics in the NAS Parallel Benchmarks. In *International Conference on Parallel and Distributed Computing and Systems*, pages 4–6, November 2002.

[17] P. Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20(1):23–52, February 1991.

[18] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies Via Critical-path Prediction. In *International Symposium on Computer Architecture*, pages 74–85, July 2001.

[19] S. Gal and B. Bachelis. An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard. *ACM Transactions on Mathematical Software*, 17(1):26–45, March 1991.

[20] E. Gibert, J. Sánchez, and A. González. Effective Instruction Scheduling Techniques for an Interleaved Cache Clustered VLIW Processor. In *International Symposium on Microarchitecture*, pages 123–133, November 2002.

[21] GNU Binutils. Free Software Foundation, 2006. `http://sources.redhat.com/binutils`.

[22] K. Goto and R. A. van de Geijn. Anatomy of High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software*, 34(12):4–29, May 2008.

[23] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. In *Design Automation Conference*, pages 726–731, June 1998.

[24] P. Gratz, K. Sankaralingam, H. Hanson, P. Shivakumar, R. McDonald, S. W. Keckler, and D. Burger. Implementation and Evaluation of a Dynamically Routed Processor Operand Network. In *International Symposium on Networks-on-Chip*, pages 7–17, May 2007.

[25] J. Hauser. SoftFloat, 2002. `http://www.jhauser.us/arithmetic/SoftFloat.html`.

[26] M. Hrishikesh, K. Farkas, N. P. Jouppi, D. Burger, S. W. Keckler, and P. Shivakumar. The Optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays. In *International Symposium on Computer Architecture*, pages 14–24, May 2002.

[27] W. Hunt, B. Maher, K. Coons, and K. S. McKinley. Optimal Huffman Tree-Height Reduction for Instruction Level Parallelism. Technical Report TR-08-34, Department of Computer Sciences, The University of Texas at Austin, 2008.

[28] D. Jiménez. Piecewise Linear Branch Prediction. In *International Symposium on Computer Architecture*, pages 382–393, June 2005.

[29] S. W. Keckler, D. Burger, M. Dahlin, L. John, C. Lin, K. McKinley, T. Keller, and K. Nowka. Tera-Op Reliable Intelligently Adaptive Processing System (TRIPS). Technical Report AFRL-IF-WP-TR-2004-1514, Air Force Research Laboratory, March 2004.

[30] J. Kepner and D. Koester. Scalable Synthetic Compact Applications, 2007. `http://www.highproductivity.org/SSCABmks.htm`.

[31] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[32] C. Kim, D. Burger, and S. W. Keckler. An Adaptive Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[33] M. Krishnan. Parallel Programming on TRIPS Using MPI. Master's thesis, Department of Computer Sciences, The University of Texas at Austin, August 2008.

[34] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a RAW Machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, October 1998.

[35] R. Lethin, A. Leung, B. Meister, P. Szilagyi, N. Vasilache, and D. Wohlford. Final Report on the The R-Stream 3.0 Compiler DARPA/AFRL Contract # F03602-03-C-0033, DTIC AFRL-RI-RS-TR-2008-160. Technical report, Reservoir Labs, Inc., May 2008.

[36] Freely Distributable LIBM. SunSoft, 2004. `http://www.netlib.org/fdlibm/readme`.

[37] B. Maher, A. Smith, D. Burger, and K. McKinley. Merging Head and Tail Duplication for Convergent Hyperblock Formation. In *International Symposium on Microarchitecture*, pages 65–76, December 2006.

[38] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers. Instruction Scheduling for a Tiled Dataflow Architecture. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 141–150, October 2006.

[39] Micron Technology, Inc., Calculating DDR Memory System Power, 2001. `http://download.micron.com/pdf/technotes/ddr/TN4603.pdf`.

[40] A. Moshovos and G. S. Sohi. Speculative Memory Cloaking and Bypassing. *International Journal of Parallel Programming*, 27(6):427–456, December 1999.

[41] R. Nagarajan, X. Chen, R. McDonald, S. W. Keckler, and D. Burger. Critical Path Analysis of the TRIPS Microprocessor. In *International Symposium on Performance Analysis of Systems and Software*, pages 104–113, April 2006.

[42] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. W. Keckler. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 74–84, September 2004.

[43] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *International Symposium on Microarchitecture*, pages 40–51, December 2001.

[44] K. Natarajan, H. Hanson, S. W. Keckler, C. R. Moore, and D. Burger. Microprocessor Pipeline Energy Analysis. In *International Symposium on Low Power Electronics and Design*, pages 282–287, August 2003.

[45] PAPI: Performance Application Programming Interface. `http://icl.cs.utk.edu/papi`.

[46] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 90–100, June 2008.

[47] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative Optimization in the Polyhedral Model: Part I, One-dimensional Time. In *International Symposium on Code Generation and Optimization*, pages 144–156, March 2007.

[48] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of Efficient Nested Loops from Polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, October 2000.

[49] R. M. Rabbah, I. Bratt, K. Asanović, and A. Agarwal. Versatility and VersaBench: A New Metric and a Benchmark Suite for Flexible Architectures. Technical Report TM-646, Laboratory for Computer Science, Massachusetts Institute of Technology, June 2004.

[50] B. Robatmili, K. E. Coons, D. Burger, and K. S. McKinley. Register Bank Assignment for Spatially Partitioned Registers. In *Workshop on Languages and Compilers for Parallel Computing*, July 2008.

[51] B. Robatmili, K. E. Coons, D. Burger, and K. S. McKinley. Strategies for Mapping Data Flow Blocks to Distributed Hardware. In *International Symposium on Microarchitecture*, November 2008.

[52] F. Roesner, D. Burger, and S. W. Keckler. Counting Dependence Predictors. In *International Symposium on Computer Architecture*, pages 215–226, June 2008.

[53] K. Sankaralingam, R. Nagarajan, S. Keckler, and D. Burger. SimpleScalar Simulation of the PowerPC Instruction Set Architecture. Technical Report TR-00-04, Department of Computer Sciences, The University of Texas at Austin, February 2001.

[54] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *International Symposium on Computer Architecture*, pages 422–433, May 2003.

[55] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *International Symposium on Microarchitecture*, pages 480–491, December 2006.

[56] S. Sethumadhavan, R. McDonald, R. Desikan, D. Burger, and S. W. Keckler. Design and Implementation of the TRIPS Primary Memory System. In *International Conference on Compute Design*, pages 470–476, October 2006.

[57] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler. Late-Binding: Enabling Unordered Load-Store Queues. In *International Symposium on Computer Architecture*, pages 347–357, June 2007.

[58] A. Seznec and P. Michaud. A Case for (Partially) TAgged GEometric History Length Branch Prediction. *Journal of Instruction-Level Parallelism*, Vol. 8, February 2006.

[59] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, September 2001.

[60] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, March 2006.

[61] A. Smith, J. Gibson, J. Burrill, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. TRIPS Intermediate Language (TIL) Manual. Technical Report TR-05-20, Department of Computer Sciences, The University of Texas at Austin, 2007.

[62] A. Smith, R. McDonald, N. Nethercote, W. Yoder, D. Burger, S. W. Keckler, and K. S. McKinley. TRIPS Application Binary Interfaced (ABI) Manual. Technical Report TR-05-22, Department of Computer Sciences, The University of Texas at Austin, 2007.

[63] Standard Performance Evaluation Corporation. `http://www.spec.org`.

[64] K. O. Stanley and R. Miikkulainen. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, March 2002.

[65] D. Stroobandt and J. V. Campenhout. Accurate Interconnection Length Estimations for Predictions Early in the Design Cycle. *VLSI Design, Special Issue on Physical Design in Deep Submicron*, 10:1–20, October 1999.

[66] D. Tarjan, S. Thoziyoor, and N. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett-Packard Labs, 2006.

[67] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar Operand Networks: On-chip Interconnect for ILP in Partitioned Architectures. In *International Symposium on High-Performance Computer Architecture*, pages 341–353, February 2003.

[68] E. Tune, D. M. Tullsen, and B. Calder. Quantifying Instruction Criticality. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 104–113, September 2002.

[69] F. von Leitner. Diet libc - A libc Optimized for Small Size, 2006. `http://www.fefe.de/dietlibc`.

[70] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: RAW Machines. *IEEE Computer*, 30(9):86–93, September 1997.

[71] B. Yoder, J. Burrill, R. McDonald, K. Bush, K. Coons, M. Gebhart, M. Govindan, B. Maher, R. Nagarajan, B. Robatmili, K. Sankaralingam, S. Sharif, A. Smith, D. Burger, S. W. Keckler, and K. S. McKinley. Software Infrastructure and Tools for the TRIPS Prototype. In *Workshop on Modeling, Benchmarking, and Simulation*, June 2007.

[72] W. Yoder, J. Gibson, J. Burrill, R. McDonald, D. Burger, S. W. Keckler, K. Sankaralingam, and R. Nagarajan. TRIPS Assembly Language (TASL) Manual. Technical Report TR-05-21, Department of Computer Sciences, The University of Texas at Austin, 2007.

[73] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. Technical Report CS-2003-05, University of Virginia, Department of Computer Science, March 2003.

List of Acronyms, Abbreviations, and Symbols

Conventional Architecture Terms

Acronym	Description
ALU:	Arithmetic Logic Unit
BTB:	Branch Target Buffer
CAM:	Content-Addressable Memory
CISC:	Complex Instruction Set Architecture
CMP:	Chip Multiprocessor
CTB:	Call Target Buffer
DGEMM:	Double-precision General Matrix Multiply (benchmark)
EEMBC:	Embedded Microprocessor Benchmark Consortium (with a standardized embedded benchmark suite)
DDR:	Dual Data Rate
DIMM:	Dual In-line Memory Module
DMA:	Direct Memory Access
DRAM:	Dynamic Random Access Memory
FLOP:	Floating-point Operation
FO4:	Fanout-of-Four: the time for an inverter to drive four copies of itself
FPC:	Floating-Point Instructions per Cycle
FPGA:	Field-Programmable Gate Array
HPCS:	High Productivity Computing Systems (a DARPA program)
ILP:	Instruction-Level Parallelism
IPC:	Instructions per Cycle
ISA:	Instruction Set Architecture
LAN:	Local Area Network
LSQ:	Load/Store Queue
MPI:	Message-Passing Interface
MPKI:	Mis-Predications per Thousand Instructions
MSHR:	Miss Status Handling Register
NOP:	No-Operation (a useless instruction)
PPC:	PowerPC
RAS:	Return Address Stack
RISC:	Reduced Instruction Set Architecture
RTL:	Register Transfer Level/Language
SDRAM:	Synchronous Dynamic Random Access Memory
SIMD:	Single Instruction Multiple Data
SMT:	Simultaneous Multithreading
SPEC:	Standard Performance Evaluation Corporation (with a standardized benchmark suite)
SRAM:	Static Random Access Memory
TLP:	Thread-Level Parallelism
VLIW:	Very Long Instruction Word (ISA)

TRIPS Architecture Terms

Acronym	Description
CLP:	Composable Lightweight Processor
EDGE:	Explicit Data Graph Execution (ISA)
NUCA:	Non-Uniform Cache Access Architecture
PCA:	Polymorphous Computing Architectures
SVM:	Streaming Virtual Machine

TFlex: Second-Generation EDGE Microarchitecture
TRIPS: Tera-op Reliable Intelligently Adaptive Processing System

TRIPS Tile and Unit Names

Acronym	Description
EBC:	External Bus Controller
DT:	Data Cache Tile
ET:	Execution Tile
GT:	Global Control Tile
IT:	Instruction Cache Tile
MT:	Memory Tile
NT:	Network Tile
RT:	Register Tile
SEQ:	Next-Block Adder
SRF:	Streaming Register File

TRIPS Network and Interface Names

Acronym	Description
C2C:	Chip-to-Chip Controller
DSN:	Data Status Network
EBI:	External Bus Interface
ESN:	External Store Network
GCN:	Global Control Network
GDN:	Global Dispatch Network
GRN:	Global Refill Network
OCN:	On-Chip Network
OPN:	Operand Network

TRIPS Software and Application Terms

Acronym	Description
GRST:	Greedy List Scheduling Register Allocation Algorithm
FLIR:	Forward-looking Infra-Red
HAL:	Hardware Abstraction Layer
NEAT:	Neuro-Evolution of Augmenting Topologies
SAR:	Synthetic Aperture Radar
SDK:	Software Development Kit
SPDI:	Static Placement, Dynamic Issue
SPS:	Spatial Path Instruction Scheduling
SSA:	Static Single Assignment
SSCA:	Scalable Synthetic Compact Application
TASL:	TRIPS Assembly Language
TIL:	TRIPS Intermediate Language
TRM:	TRIPS Resource Manager
VDG:	Value Dependence Graph